Data Intelligence Agents: Interpreting, Modeling, and Querying Enterprise Data via Autonomous Coding Agents

Anoushka Vyas **Aarushi Dhanuka** **Sina Khoshfetrat Pakazad** **Henrik Ohlsson**
C3 AI
{anoushka.vyas, aarushi.dhanuka, sina.pakazad, henrik.ohlsson}@c3.ai

Abstract

Production data integration is bottlenecked by repeated, lossy handoffs between data owners, engineers, and analysts who must collaboratively discover, structure, and query enterprise data. We present **D**ata **I**ntelligence **A**gents (DIA), a system of three agents (*Data Interpreter*, *Schema Creator*, and *Query Generator*) that compresses this workflow by treating autonomous coding agents (ACAs) as a first-class abstraction: rather than emitting text, the agents generate, execute, validate, and repair concrete artifacts, draw on a shared memory for experience reuse, and surface each for review by domain experts. DIA is deployed in production for enterprise customers. We study the *Query Generator* in depth and evaluate it in fully autonomous mode across seven SQL benchmarks spanning four task categories and four dialects. It matches or surpasses the best published results on all seven, demonstrating that an architecture grounded in execution, built on ACAs and a shared memory, generalizes across the data intelligence workload with adaptation confined to natural-language instructions.

1 Introduction

Enterprise data work rarely fails for lack of data; it fails because raw data must be discovered, understood, structured, and queried before it can support analysis. In practice this passes through repeated, lossy handoffs between the data owners who understand what fields mean, the engineers who structure and validate the data, and the analysts who query it. Each handoff loses context and adds latency: a misread field or an implicit business rule forces schema changes, pipeline rework, and query rewrites. The opportunity is to keep the domain experts who understand the data in control while compressing the engineering cycle.

Large language models make each step look tractable in isolation, yet existing systems address fragments of this workflow rather than closing it.

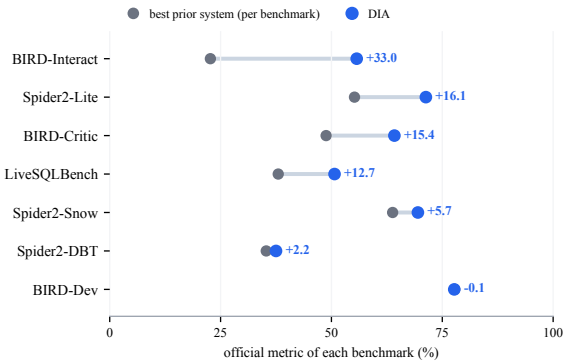

Figure 1: DIA against the best prior system on each of the seven SQL benchmarks, ordered by margin. Each benchmark is scored by its official metric.

Pipeline systems for text-to-SQL chain handcrafted modules, each tuned for one subtask and brittle when the task changes (Pourreza and Rafiei, 2023; Pourreza et al., 2025). Specialists trained with reinforcement learning reach high accuracy on a single benchmark but are locked to one dialect and need costly retraining per variant (Yang et al., 2025; Li et al., 2025a). Agentic explorers probe the database live but keep no memory across sessions, restarting from scratch on every query (Cao et al., 2026; Deng et al., 2025). SQL agents with persistent memory store and replay past experience, but keep a single store and a narrow evaluation (Biswal et al., 2026; Yang et al., 2026; Chu et al., 2024; Chen et al., 2025). Across these approaches the system emits text (queries or critiques) rather than the executable, inspectable artifacts that enterprise data work consumes, and none addresses the upstream understanding and schema construction stages that decide whether the resulting SQL has anything sensible to run against.

DIA closes this loop. It directs a single ACA (a coder driven by an LLM in a sandboxed environment) across the discovery, schema construction, and query stages (Figure 2), surfacing each artifact

EMNLP 2026 Industry Track, pages 2500–2531
October 24-29, 2026 ©2026 Association for Computational Linguistics

for review by domain experts (Wang et al., 2024b; Song et al., 2026). The key contributions are as follows:

1. The first system, to our knowledge, to treat the ACA rather than the LLM as the central abstraction for data intelligence: three agents (*Data Interpreter*, *Schema Creator*, and *Query Generator*) realized as a single ACA over a shared workspace that turns raw enterprise data into validated, queryable schemas and grounded answers, replacing lossy text handoffs with inspectable artifacts.

2. The design of the *Query Generator*: a single generalist agent that handles SQL generation, debugging, conversational interaction, and project completion across four dialects through self-correction grounded in execution and a shared memory for experience reuse, with adaptation confined to natural-language instructions.

3. A broad empirical study: in fully autonomous mode on seven SQL benchmarks spanning four task categories and four dialects, with a single LLM and no fine-tuning, the *Query Generator* matches or surpasses the best published results on all seven (Figure 1).

2 Related Work

Text-to-SQL systems. Text-to-SQL is a large and active area (Hong et al., 2025), but the work fragments by task setting. Most systems target single-shot query generation and improve accuracy on it through multi-agent collaboration (MAC-SQL (Wang et al., 2023), CHESS (Talaei et al., 2024)), ensemble pipelines (OpenSearch-SQL (Xie et al., 2025), XiYan-SQL (Gao et al., 2024)), or component-level advances in schema linking (Pradeep et al., 2025) and decoding (Sharma et al., 2025). The pipeline, reinforcement-learning, and agentic systems of Section 1 address this same setting. Other settings are served by separate, specialized systems: conversational querying (SParC (Yu et al., 2019b), CoSQL (Yu et al., 2019a)) and declarative querying over heterogeneous data (Khabiri et al., 2025). The concurrent AgentNLQ (Bogdanov et al., 2026) is the closest, a multi-agent system for general-purpose NL-to-SQL, but it too addresses a single setting. DIA's *Query Generator* instead spans four task categories across four dialects with one agent.

Data understanding and schema generation. DIA is a system, not a single SQL model, and its other two agents build on work done so far in isolation. For data understanding, large tabular models profile tables and infer column semantics, types, and relationships (TableGPT2 (Su et al., 2024)). For schema construction, recent agents build relational schemas from natural language (Text2Schema (Wang et al., 2025b)) and prepare raw data for analysis (DeepPrep (Fan et al., 2026)); new benchmarks measure data agents across the full data intelligence lifecycle, from engineering to analysis (DAComp (Lei et al., 2026)). These are standalone tools and evaluations. DIA's *Data Interpreter* and *Schema Creator* instead work as agents in one system, handing validated, executable artifacts to the *Query Generator*.

Generalist and memory agents. DIA inherits the generalist coding agent paradigm, where one agent solves many tasks through code execution instead of separate modules for each task (OpenHands-Versa (Soni et al., 2025), CodeAct (Wang et al., 2024b)). It treats ACAs as a first-class abstraction, as does NEMO (Song et al., 2026) for optimization modeling. DIA also draws on agents that learn from experience: ARIA (He et al., 2025) keeps a knowledge repository that improves over time, and Voyager (Wang et al., 2024a), Reflexion (Shinn et al., 2023), and ReasoningBank (Ouyang et al., 2025) accumulate skills, reflections, or reasoning strategies. These build experience for one agent. DIA's three agents instead share a memory and reuse experience across the system.

3 Methodology

3.1 Overview

A central abstraction in DIA is remote interaction with an ACA, an execution-capable counterpart to a text-only model call. Operating within a sandboxed environment, the ACA generates, executes, inspects, and revises code, so that every output is an executable artifact and admits execution-aware validation (Song et al., 2026; Wang et al., 2024b). DIA drives the ACA with natural-language instructions and references to existing workspace artifacts, and receives code, execution traces, and results in return.

DIA is a system of three agents, realized as a single ACA invoked over a shared workspace W: a *Data Interpreter* that profiles raw sources into a structured interpretation (Section 3.3); a *Schema*

Creator that materializes and validates a relational database from that interpretation (Section 3.4); and a *Query Generator* that translates natural-language questions into executed SQL (Section 3.5). The agents communicate through W, in which artifacts persist as files, rather than by exchanging text, and each draws on a shared memory M for experience reuse (Section 3.2); we write $M^* \subseteq M$ for the subset retrieved at each invocation. Each artifact is surfaced for review by domain experts.

3.2 Memory

Memory in DIA is artifact-based: because the ACA works in a sandbox, what it carries forward is the concrete artifacts it has produced and validated, schemas, loading and transformation scripts, validation reports, query logs, and prior solutions, rather than textual summaries of them. Within a task, the agents build on the artifacts already in the workspace W (Section 3.1), each consuming what the previous produced. Across tasks, an experience store M retains a reusable subset in three tiers that mirror a memory hierarchy: retrieved examples, an episodic tier of similar past question-and-solution pairs surfaced for the current question; session lessons, conditional rules confirmed on the current database; and cross-session lessons, the long-term semantic subset of those rules that generalize across databases.

Memory is pull-based and verified before use. Items are surfaced only by reference, and the agent reads a body only when it judges it relevant. Nothing changes an answer until a live probe confirms its precondition on the current data, so stale experience is caught by execution rather than propagated. The agent writes memory itself: after answering it reflects and records a conditional rule with its evidence, updating the store

$$M \leftarrow w(M, a, o),$$

where a is the artifact just produced and o its observed outcome, and w admits a cross-session lesson only when o bears it out, with no learned or human judge, keeping the mechanism training-free. All three agents share one memory and consume M^* through their signatures.

3.3 Data Interpreter

Given a collection of heterogeneous raw sources $D = \{d_1, \ldots, d_n\}$ such as CSV, JSON, and Excel files, the *Data Interpreter* writes and executes profiling code that produces a structured interpretation P:

$$\mathcal{I} : (D, M^*) \rightarrow P.$$

Rather than describe D in natural language, the ACA derives P entirely from executed code, and the resulting artifact is what downstream agents consume. P records, for each source, the inferred schema with column names and semantic types; per-column value distributions, null statistics, and pattern observations; candidate primary and foreign keys; likely join paths across sources; and data-quality observations that require intervention.

3.4 Schema Creator

Given the interpretation P and the raw sources D, the *Schema Creator* generates and executes loading and validation code that materializes a working database:

$$\mathcal{S} : (P, D, M^*) \rightarrow (\Sigma, \beta),$$

where $\Sigma = (T, K, C)$ specifies the tables T with their columns and types, the key constraints K (each table's primary key and the foreign keys linking tables), and the integrity constraints C (column-level rules the data must satisfy, such as not-null and value-range checks), and β is Σ instantiated as physical tables and populated with the records in D. The ACA works under a load-first-normalize-second discipline: staging tables ingest every record from D with provenance fields recording source file and load timestamp; refined tables and views apply typing and structure on top. The ACA then validates (Σ, β) along four axes: (i) row-count reconciliation between sources and β; (ii) column coverage, requiring every source column to be carried through and any rename to be recorded; (iii) key validity, checking primary-key uniqueness and foreign-key referential integrity; and (iv) load integrity, routing records that cannot be ingested to per-table reject buffers rather than dropping them silently. A set of test queries τ is executed against β; the schema is accepted only when ingestion succeeds and every query in τ executes as expected. Alongside β, the ACA emits a schema manifest enumerating T, K, and column mappings, and a validation report summarizing the four checks.

3.5 Query Generator

Given a natural-language question q and the database (Σ, β), the *Query Generator* writes and executes SQL to answer it:

$$\mathcal{Q} : (q, \Sigma, \beta, M^*) \rightarrow y,$$

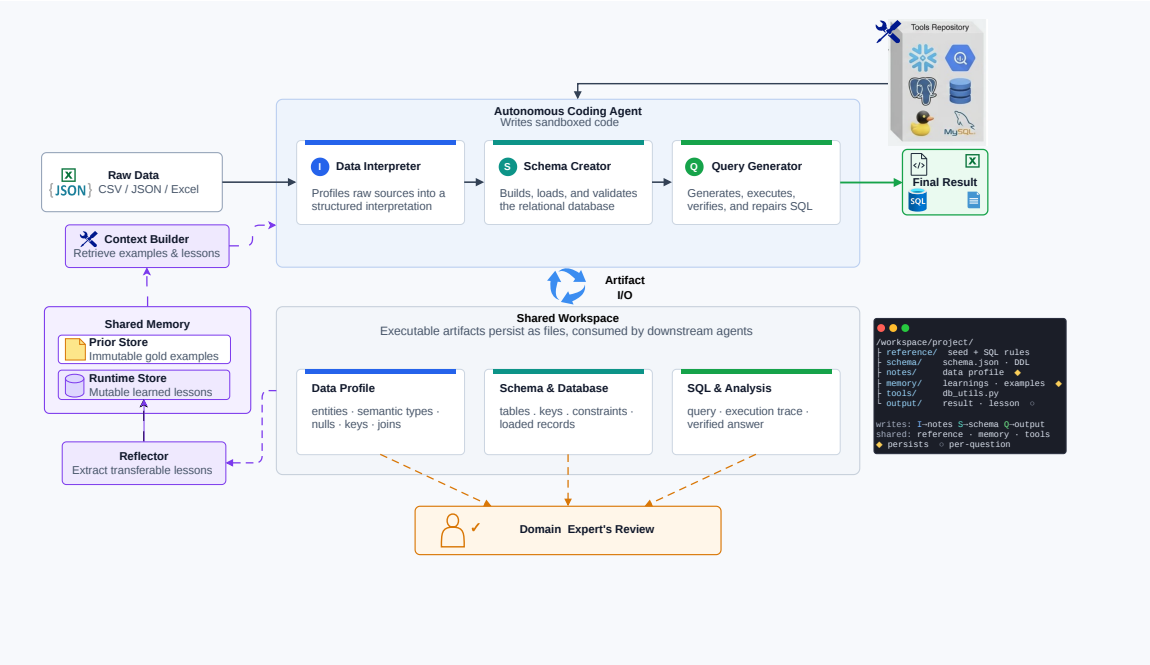

Figure 2: The DIA system. A single ACA operating over a shared workspace W realizes three agents (*Data Interpreter*, *Schema Creator*, and *Query Generator*), turning raw data D and a question q into a grounded answer R. Each agent reads and writes executable artifacts in W; all draw on a shared memory M; domain experts review each artifact.

where y is a SELECT statement for analytical questions or a DDL/DML statement for modification tasks. The ACA accesses β in read-only mode for analytical questions. Generation is execution-grounded and proceeds in four phases, with the ACA writing and executing SQL throughout.

Shape declaration. Before generating y, the ACA derives from q an expected result shape

$$\kappa = (C_\kappa,\ g_\kappa,\ o_\kappa,\ f_\kappa),$$

where C_κ is the column list implied by q, g_κ the row granularity (one row per entity, per group, per time bucket, and so on), o_κ the ordering specification, and f_κ the filter conjunction extracted from q. For modification tasks, κ specifies the target objects and the intended post-condition on β.

Schema exploration. The ACA executes lightweight probe queries against Σ and β to confirm that join keys exist, sample representative column values to fix their format, and verify cardinality assumptions, rather than inferring structure from column names alone.

Generation and execution. The ACA produces a candidate query

$$y = \mathcal{G}(q, \Sigma, M^*, \kappa)$$

conditioned on the question, schema, retrieved memory, and declared shape, and executes it to obtain $R = \mathrm{exec}(y, \beta)$.

Self-verification. The agent does not treat the first query it writes as final. It checks the result against the declared shape with its own indicator

$$V(R, \kappa) = \begin{cases} 1 & \text{if } R \text{ satisfies } \kappa, \\ 0 & \text{otherwise,} \end{cases}$$

evaluated componentwise against C_κ, g_κ, o_κ, and f_κ. For modification tasks, V checks that the intended post-condition holds in β. This check is the agent's own and is computed from execution rather than supplied by an external verifier or human: when $V(R, \kappa) = 0$, the agent diagnoses the gap, revises y, and re-executes within the same pass before emitting an answer. The procedure is independent of task category and SQL dialect; only the grammar of κ varies across them.

4 Evaluation

4.1 Setup

We evaluate the *Query Generator* on seven public SQL benchmarks: BIRD-Dev (Li et al., 2023), BIRD-Critic (Li et al., 2025b), LiveSQLBench (BIRD-bench Team, 2025), BIRD-Interact (Huo

et al., 2026), and the Spider2 family (Lei et al., 2025) (Spider2-Lite, Spider2-Snow, and Spider2-DBT). Together they comprise 4,187 instances spanning four task categories and four SQL dialects: generation (BIRD-Dev, LiveSQLBench, Spider2-Lite, Spider2-Snow), debugging (BIRD-Critic), conversational interaction (BIRD-Interact), and dbt project completion (Spider2-DBT), across SQLite, PostgreSQL, Snowflake, and DuckDB. Several benchmarks contain finer task categories, such as data modification in LiveSQLBench and personalization in BIRD-Critic, which we break down in Appendix B; full dataset details are given in Appendix A.

All experiments use a unified system configuration: OpenHands, powered by Claude Sonnet 4.5 with no fine-tuning, acts as the ACA, while o3 serves as the user simulator in BIRD-Interact's conversational protocol. Customization for each benchmark is confined to a standing seed file and the per-question prompt scaffolding (Appendix E), and every run is fully autonomous with no human intervention. Implementation details are given in Appendix I. The shared memory examined in Section 4.5 retrieves its episodic examples from the BIRD train split, indexed offline and disjoint from the BIRD-Dev evaluation set, and is detailed in Appendix F.

For each benchmark we compare against the best available prior results (Appendix H). Systems with an accompanying publication are cited in Table 1, and the remainder are listed by name. The primary metric is each benchmark's official one: execution accuracy throughout, except task success rate on BIRD-Interact and database-match accuracy on Spider2-DBT. Additional official metrics and their definitions are given in Appendices B and A. Reported execution accuracy on BIRD-Dev varies by a small margin across works, owing to differences in evaluation harnesses and to noise and periodic corrections in the benchmark's gold queries (Wretblad et al., 2024). We release granular per-instance results via HuggingFace[1].

4.2 Main results

Table 1 summarizes performance across the seven benchmarks, comparing the *Query Generator* against the strongest available agent-based and training-based systems. Overall, it achieves strong and consistent performance, matching or surpassing the best published result on all seven benchmarks, by large margins on several.

The gains are largest on the tasks where prior systems struggle most: +33.0 points on BIRD-Interact (conversational interaction), +16.1 on Spider2-Lite, +15.4 on BIRD-Critic (debugging), and +12.7 on LiveSQLBench. Appendix D traces where the BIRD-Interact margin comes from, with worked interaction traces and an interaction-time scaling analysis. On the most competitive benchmarks the lead narrows but holds: +5.7 on Spider2-Snow and +2.2 on Spider2-DBT, where the agent edits a real dbt repository rather than emitting a single query and the strongest prior system is built on GPT-5.4. On BIRD-Dev, the most saturated benchmark, where the field is clustered within a point, DIA is level with the strongest published result, MARS-SQL (Yang et al., 2025), an RL-trained specialist (77.7 vs. 77.8). One model and scaffold thus serve four task categories and four dialects, with adaptation confined to natural-language standing instructions.

The per-category breakdown (Appendix B) shows two consistent patterns beneath the headline scores. DIA is strongest on the more structured task variants: BIRD-Critic Management (78.7) and LiveSQLBench Modification (66.3) both exceed the same benchmark's pure-query slice, because modification tasks reward the agent's habit of declaring the target object and validating it by execution before answering. It is weakest on high-level questions that name a composite metric without spelling out its formula (41.6 on LiveSQLBench, 47.6 on BIRD-Interact), where the agent must either decompose the metric or ask. On BIRD-Interact, the phase-2 conditional pass rate of 86.8 shows that once a correct phase-1 query lands, follow-ups are almost always answered correctly.

4.3 Component ablation

To separate the contribution of the system from that of the underlying model, we ablate one component at a time on the full BIRD-Dev set (1,534 questions), holding the model and the agent loop fixed (Table 2). Removing shape declaration and removing execution-based self-verification each cost 7.5 points; these are two halves of one output-contract discipline (Appendix E), so disabling either one removes it. Removing the shared memory costs 2.6 points. Reducing the system to a bare OpenHands agent on the same model, given only the

[1] https://huggingface.co/datasets/c3aiia3c/dia-emnlp2026

Benchmark	Dialect	Task	System	Score (%)
BIRD-Dev	SQLite	Generation	Agentar-Scale-SQL (Wang et al., 2025a)	74.9
			CHASE-SQL (Pourreza et al., 2025)	74.9
			DIA	77.7
			MARS-SQL (Yang et al., 2025)	**77.8**
BIRD-Critic	SQLite	Debugging	Claude Opus 4.6	46.2
			BIRD-Talon-14B	48.0
			Gemini 3.1 Pro Preview	48.8
			DIA	**64.2**
LiveSQLBench	PostgreSQL	Generation	OpenHands + Kimi 2.5	32.2
			OpenHands + Claude Sonnet 4.5	35.2
			OpenHands + Claude Opus 4.6	38.0
			DIA	**50.7**
BIRD-Interact	PostgreSQL	Conversational	Claude Opus 4.6	17.5
			MERIT (Wang et al., 2026) + Claude Opus 4.6	20.8
			MERIT (Wang et al., 2026) + GPT-5.4	22.7
			DIA	**55.7**
Spider2-Lite	Snowflake, SQLite	Generation	DSR-SQL (Hao et al., 2025)	46.8
			AutoLink (Wang et al., 2025c) + DeepSeek-R1	52.3
			ReFoRCE (Deng et al., 2025) + o3	55.2
			DIA	**71.3**
Spider2-Snow	Snowflake	Generation	APEX-SQL (Cao et al., 2026)	53.0
			ReFoRCE (Deng et al., 2025) + o3	62.9
			DSR-SQL (Hao et al., 2025) + DeepSeek-R1	63.8
			DIA	**69.5**
Spider2-DBT	DuckDB	Project completion	Spider-Agent (Lei et al., 2025) + o1-preview	13.2
			Spider-Agent + Claude 3.7 Sonnet	14.7
			Spider-Agent-DBT + GPT-5.4	35.3
			DIA	**37.5**

Table 1: DIA against the strongest published baselines across seven SQL benchmarks. Score is each benchmark's official primary metric (higher is better): execution accuracy in all cases except task success rate (BIRD-Interact) and database-match accuracy (Spider2-DBT). Each benchmark's primary and additional metrics are described in detail in Appendix A.

schema and the question, costs 14.1 points, a second system-versus-model control that mirrors the LiveSQLBench gap in Table 1 (35.2 to 50.7). The clarification policy is evaluated on BIRD-Interact, whose interactive protocol depends on it: removing it drops task success from 55.7 to 14.5, a 41.2-point fall (Appendix D). Across benchmarks, then, it is these components rather than the backbone that drive the gains.

This ablation also maps design to benchmark gains. Shape declaration and self-verification, the output-contract discipline ablated above, underpin the debugging and modification gains on BIRD-Critic (+15.4) and LiveSQLBench (+12.7) in Table 1, since both tasks reward validating a candidate object or edit against the database before returning it. The clarification policy accounts for most of the +33.0-point margin on BIRD-Interact, whose conversational protocol rewards resolving ambiguity rather than guessing. Shared memory contributes more evenly across benchmarks, since the rules it accumulates concern joins, aggregation, and output convention, failure modes that recur regardless of task type (Section 4.5).

4.4 Cost, latency, and variance

We report per-question cost and latency for DIA on BIRD-Dev, measured over the full run (Table 3). A question costs about USD 0.10 on average, ranging from 0.03 to 0.20 across databases, and processes about 145k input tokens (of which roughly 94% are served from cache) and about 1.6k output tokens. End-to-end runtime has a median of 38 seconds; the 90th percentile, meaning all but the slowest one in ten questions, is 68 seconds. Since questions are independent, throughput scales with parallel workers.

To quantify run-to-run variability under the non-determinism of the underlying agent, we run BIRD-Dev three times, obtaining 77.7, 76.1, and 77.0, a mean of 76.9 with a standard deviation of 0.8 (Table 2). The 0.1-point gap to the strongest published

Configuration	Score (%)	Δ
Full DIA	77.7	—
No shared memory	75.1	–2.6
No self-verification	70.2	–7.5
No shape declaration	70.2	–7.5
Bare OpenHands agent	63.6	–14.1
Full DIA, mean of 3 runs	76.9 ± 0.8	—

Table 2: Component ablation on the full BIRD-Dev set (1,534 questions), removing one component at a time on the same model and instances, plus the mean and standard deviation across three independent full runs. The clarification policy is ablated separately on BIRD-Interact, dropping from 55.7 to 14.5, a fall of 41.2 points, whose protocol requires it.

Metric	Mean
Cost / question	USD 0.10 (range 0.03–0.20)
Input tokens / question	145k (94% cached)
Output tokens / question	1.6k
Runtime, median	38 s
Runtime, 90th percentile	68 s

Table 3: Per-question cost, token usage, and latency for DIA on BIRD-Dev, averaged over the full run; the cost range is across databases. Runtime is the distribution over questions; the 90th percentile bounds all but the slowest one in ten.

result in Table 1 is thus well within run-to-run noise, and within the documented annotation noise of the benchmark (Wretblad et al., 2024).

4.5 Learned Rules

On BIRD-Dev, for example, DIA accumulates a small store of conditional rules of a few recurring kinds, each interpretable and grounded in a check on the data. This accumulation introduces no test leakage: the agent never sees gold answers or a grading signal, every rule is distilled from its own execution observations and re-verified on the live database before it can change an answer, and the episodic examples it retrieves come from the disjoint BIRD train split. Rules form in two stages: a rule begins as a concrete within-database observation and is promoted to a cross-database rule only when later questions bear it out. On `california_schools`, for instance, the agent counting schools through a one-to-many join saw `COUNT(*)` return 9,977 for one school (its fact-row count) but `COUNT(DISTINCT CDSCode)` return 1. The rule promoted from this is to count entities through a repeating join with `COUNT(DISTINCT pk)`, not `COUNT(*)`. The rule kinds it records span joins, aggregation, and output convention, the same recurring failure structures the error analysis (Section 4.6) finds behind most wrong answers. Appendix F gives representative rules with their evidence, both stages of this promotion, the full three-tier store, and a worked trace of a session lesson redirecting a later answer.

4.6 Error analysis

Almost every failed instance runs to completion and returns a wrong answer rather than failing to execute, so the remaining errors are overwhelmingly semantic rather than syntactic. Appendix C sorts them into three recurring classes: reasoning, output convention, and grounding, with reasoning by far the most frequent. A reasoning failure executes cleanly but answers a subtly different question, most often a wrong join, filter, or formula on an under-specified request.

5 Conclusion

We presented DIA, which treats the ACA as the central abstraction for enterprise data intelligence: a single ACA over a shared workspace, invoked as three agents, produces executable artifacts that domain experts review rather than text they must trust. The premise is that one agent grounded in execution can replace a family of specialized systems, and our evaluation supports it: with a single LLM and no fine-tuning, the *Query Generator* matches or surpasses the best published results on all seven benchmarks (Table 1), across task categories and dialects that prior work covers with separate, task-specific systems.

Two lessons recur across benchmarking and deployment alike. Execution grounding helps most where an answer can be checked against the database, and least on upstream domain ambiguity that execution cannot resolve; and for a system meant to be reviewed rather than trusted outright, the design choice that matters most is emitting inspectable artifacts rather than natural-language explanations of them. DIA is deployed in production for enterprise customers (Appendix G), where these same properties, execution-grounded artifacts, human review, and permission-scoped access, provide the auditability such settings require.

6 Limitations and Future Work

DIA trades computation for reliability. Each question runs through an iterative loop of generation, execution, and verification, with conversational tasks adding multi-turn interaction, so mean time per question ranges from under a minute on single-shot generation tasks to roughly ten minutes on multi-turn conversational tasks; Section 4.4 reports the full BIRD-Dev cost and latency distribution. This is acceptable when answers feed durable artifacts but may be prohibitive for interactive or high-throughput use. Caching exploration, parallelizing independent runs, and distilling routine patterns into cheaper components would reduce it.

Verification in DIA is execution-grounded but not semantic. The agent checks an executed result against the result shape it has itself derived from the question, so when it misreads intent, the query and the check inherit the same misreading and a wrong answer passes. The residual failures in Section 4.6 concentrate there: join, filter, and formula reasoning on under-specified questions. Engaging the semantics of the question, rather than the shape of its answer, is the clearest path to closing the gap.

Finally, the evaluation is broad in tasks and dialects but deliberately narrow elsewhere: one of the three agents, a single backbone, a simulated rather than human user on conversational tasks, and memory examined only qualitatively. All results use one LLM (Claude Sonnet 4.5) with no fine-tuning, so we do not yet establish that the gains transfer across backbones; the ablation in Section 4.3 bounds the single-model risk, since DIA's margin over a bare agent on the same model and substrate (14.1 points) indicates the system, not the model, carries the result, but a study across backbones remains future work. We plan to widen each dimension: benchmarking the *Data Interpreter* and *Schema Creator* on data preparation and schema generation, testing sensitivity across LLMs such as GPT and Gemini, and studying real users. Memory is a further direction: experience accumulates as artifacts and rules in the workspace, and organizing it with graph-structured links rather than as files would let the system mine that experience more effectively.

References

BIRD-bench Team. 2025. LiveSQLBench: A contamination-free, continuously-evolving text-to-SQL benchmark. https://livesqlbench.ai.

Asim Biswal, Chuan Lei, Xiao Qin, Aodong Li, Balakrishnan Narayanaswamy, and Tim Kraska. 2026. AgentSM: Semantic memory for agentic text-to-SQL. *arXiv preprint arXiv:2601.15709*.

Olena Bogdanov, Yeunji Jung, Chandra Dhir, Pareekshitreddy Gaddam, Saurabh Jain, Lakshmi Tumati, Vijay Parthasarathy, and Anup Shirgaonkar. 2026. AgentNLQ: A general-purpose agent for natural language to SQL. *arXiv preprint arXiv:2605.19010*.

Bowen Cao, Weibin Liao, Yushi Sun, Dong Fang, Haitao Li, and Wai Lam. 2026. APEX-SQL: Talking to the data via agentic exploration for text-to-SQL. *arXiv preprint arXiv:2602.16720*.

Zui Chen, Han Li, Xinhao Zhang, Xiaoyu Chen, Chunyin Dong, Yifeng Wang, Xin Cai, Su Zhang, Ziqi Li, Chi Ding, Jinxu Li, Shuai Wang, Dousheng Zhao, Sanhai Gao, and Guangyi Liu. 2025. RubikSQL: Lifelong learning agentic knowledge base as an industrial NL2SQL system. *arXiv preprint arXiv:2508.17590*.

Zhibo Chu, Zichong Wang, and Qitao Qin. 2024. Leveraging prior experience: An expandable auxiliary knowledge base for text-to-SQL. *arXiv preprint arXiv:2411.13244*.

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

Meihao Fan, Ju Fan, Yuxin Zhang, Shaolei Zhang, Xiaoyong Du, Jie Song, Peng Li, Fuxin Jiang, Tieying Zhang, and Jianjun Chen. 2026. DeepPrep: An LLM-powered agentic system for autonomous data preparation. *arXiv preprint arXiv:2602.07371*.

Yingqi Gao, Yifu Liu, Xiaoxia Li, Xiaorong Shi, Yin Zhu, Yiming Wang, Shiqi Li, Wei Li, Yuntao Hong, Zhiling Luo, and 1 others. 2024. A preview of XiYan-SQL: A multi-generator ensemble framework for text-to-SQL. *arXiv preprint arXiv:2411.08599*.

Zhifeng Hao, Qibin Song, Ruichu Cai, Boyan Xu, and 1 others. 2025. Text-to-SQL as dual-state reasoning: Integrating adaptive context and progressive generation. *arXiv preprint arXiv:2511.21402*.

Yufei He, Ruoyu Li, Alex Chen, Yue Liu, Yulin Chen, Yuan Sui, Cheng Chen, Yi Zhu, Luca Luo, Frank Yang, and Bryan Hooi. 2025. Enabling self-improving agents to learn at test time with human-in-the-loop guidance. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1625–1653. Association for Computational Linguistics.

Zijin Hong, Zheng Yuan, Qinggang Zhang, Hao Chen, Junnan Dong, Feiran Huang, and Xiao Huang. 2025. Next-generation database interfaces: A survey of LLM-based text-to-SQL. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*.

Nan Huo, Xiaohan Xu, Jinyang Li, Per Jacobsson, Shipei Lin, and 1 others. 2026. BIRD-INTERACT: Re-imagining text-to-SQL evaluation for large language models via lens of dynamic interactions. In *International Conference on Learning Representations (ICLR)*.

Elham Khabiri, Jeffrey O. Kephart, Fenno F. Heath, Srideepika Jayaraman, Yingjie Li, Fateh A. Tipu, Dhruv Shah, Achille Fokoue, and Anu Bhamidipaty. 2025. Declarative techniques for NL queries over heterogeneous data. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1744–1761. Association for Computational Linguistics.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, and 1 others. 2025. Spider 2.0: Evaluating language models on real-world enterprise text-to-SQL workflows. In *International Conference on Learning Representations (ICLR)*.

Fangyu Lei, Jinxiang Meng, Yiming Huang, Junjie Zhao, Yitong Zhang, Jianwen Luo, Xin Zou, Ruiyi Yang, Wenbo Shi, Yan Gao, Shizhu He, Zuo Wang, Qian Liu, Yang Wang, Ke Wang, Jun Zhao, and Kang Liu. 2026. DAComp: Benchmarking data agents across the full data intelligence lifecycle. In *International Conference on Learning Representations (ICLR)*.

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. 2025a. OmniSQL: Synthesizing high-quality text-to-SQL data at scale. *Proceedings of the VLDB Endowment*.

Jinyang Li, Binyuan Hui, Ge Qu, and 1 others. 2023. 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)*.

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, and 1 others. 2025b. SWE-SQL: Illuminating LLM pathways to solve user SQL issues in real-world applications. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. 2025. ReasoningBank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed in-context learning of text-to-SQL with self-correction. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Mohammadreza Pourreza, Shayan Talaei, Ruoxi Sun, Xingchen Wang, Shuaichen Zhang, Azalia Mirhoseini, Amin Saberi, and Sercan O Arik. 2025. CHASE-SQL: Multi-path reasoning and preference optimized candidate selection in text-to-SQL. In *International Conference on Learning Representations (ICLR)*.

Kiran Pradeep, Kirushikesh Db, Nishtha Madaan, Sameep Mehta, and Pushpak Bhattacharyya. 2025. Divide, link, and conquer: Recall-oriented schema linking for NL-to-SQL via question decomposition. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1727–1743. Association for Computational Linguistics.

Chetan Sharma, Ramasuri Narayanam, Soumyabrata Pal, Kalidas Yeturu, Shiv Kumar Saini, and Koyel Mukherjee. 2025. TTD-SQL: Tree-guided token decoding for efficient and schema-aware SQL generation. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1287–1298. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Yang Song, Anoushka Vyas, Zirui Wei, Sina Khoshfetrat Pakazad, Henrik Ohlsson, and Graham Neubig. 2026. NEMO: Execution-aware optimization modeling via autonomous coding agents. In *International Conference on Machine Learning (ICML)*.

Aditya Bharat Soni, Boxuan Li, Xingyao Wang, Valerie Chen, and Graham Neubig. 2025. Coding agents with multimodal browsing are generalist problem solvers. *arXiv preprint arXiv:2506.03011*.

Aofeng Su, Aowen Wang, Chao Ye, Chen Zhou, Ga Zhang, Gang Chen, Guangcheng Zhu, Haobo Wang, Haokai Xu, Hao Chen, and 1 others. 2024. TableGPT2: A large multimodal model with tabular data integration. *arXiv preprint arXiv:2411.02059*.

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

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, and Xing Sun. 2023. MAC-SQL: A multi-agent collaborative framework for text-to-SQL. *arXiv preprint arXiv:2312.11242*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2024a. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research (TMLR)*.

Pengfei Wang, Baolin Sun, Xuemei Dong, Yaxun Dai, Hongwei Yuan, and 1 others. 2025a. Agentar-Scale-SQL: Advancing text-to-SQL through orchestrated test-time scaling. *arXiv preprint arXiv:2509.24403*.

Qin Wang, Youhuan Li, Yansong Feng, Si Chen, Ziming Li, Pan Zhang, Zihui Si, Yixuan Chen, Zhichao Shi, Zebin Huang, Guo Chen, and Wenqiang Jin. 2025b. Text2Schema: Filling the gap in designing database table structures based on natural language. *arXiv preprint arXiv:2503.23886*.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024b. Executable code actions elicit better LLM agents. In *International Conference on Machine Learning (ICML)*.

Yibo Wang, Nikki Lijing Kuang, Philip S. Yu, Zhewei Yao, and Yuxiong He. 2026. Learning to retrieve: Dual-level long-term memory for text-to-SQL agents. *arXiv preprint arXiv:2606.00547*.

Ziyang Wang, Yuanlei Zheng, Zhenbiao Cao, Xiaojin Zhang, Zhongyu Wei, and 1 others. 2025c. AutoLink: Autonomous schema exploration and expansion for scalable schema linking in text-to-SQL at scale. *arXiv preprint arXiv:2511.17190*.

Niklas Wretblad, Fredrik Riseby, Rahul Biswas, Amin Ahmadi, and Oskar Holmström. 2024. 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, Bangkok, Thailand. Association for Computational Linguistics.

Xiangjin Xie, Guangwei Xu, Lingyan Zhao, and Ruijie Guo. 2025. OpenSearch-SQL: Enhancing text-to-SQL with dynamic few-shot and consistency alignment. *arXiv preprint arXiv:2502.14913*.

Haolin Yang, Jipeng Zhang, Zhitao He, Alexander Zhou, and Yi R. Fung. 2025. MARS-SQL: A multi-agent reinforcement learning framework for text-to-SQL. *arXiv preprint arXiv:2511.01008*.

Zerui Yang, Weichuan Wang, Yanwei Xu, Linqi Song, Yudai Matsuda, Wei Han, and Bo Bai. 2026. Memo-SQL: Structured decomposition and experience-driven self-correction for training-free NL2SQL. *arXiv preprint arXiv:2601.10011*.

Tao Yu, Rui Zhang, Heyang Er, Suyi Li, Eric Xue, Bo Pang, and 1 others. 2019a. CoSQL: A conversational text-to-SQL challenge towards cross-domain natural language interfaces to databases. In *Conference on Empirical Methods in Natural Language Processing (EMNLP-IJCNLP)*.

Tao Yu, Rui Zhang, Michihiro Yasunaga, Yi Chern Tan, Xi Victoria Lin, and 1 others. 2019b. SParC: Cross-domain semantic parsing in context. In *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Appendix Contents

A Datasets **12**
- A.1 Descriptions . . . 12
- A.2 Metrics . . . 13

B Additional Results **14**
- B.1 Category and phase breakdowns . . . 14
- B.2 Difficulty and dialect breakdowns . . . 15
- B.3 Per-database results . . . 15

C Error Analysis **17**
- C.1 Failure classes across benchmarks . . . 18
- C.2 Per-benchmark patterns . . . 19
- C.3 Interaction failure behaviour . . . 19

D Case Studies **20**
- D.1 The BIRD-Interact protocol . . . 20
- D.2 A passing trace . . . 21
- D.3 A stuck-loop failure . . . 22
- D.4 A Phase-2 cascade . . . 23
- D.5 Interaction policies . . . 24

E Standing Instructions **25**
- E.1 The workflow skeleton . . . 25
- E.2 A worked debugging example . . . 26

F Memory: Store and Contents **27**
- F.1 The three tiers . . . 27
- F.2 Representative rules . . . 27
- F.3 Episodic-to-semantic generalization . . . 28
- F.4 A learned rule in use . . . 28

G Production Deployment **29**

H Leaderboard Reference **31**

I Configuration **31**

A Datasets

A.1 Descriptions

Table 4 summarizes the seven benchmarks. All are public, and we evaluate their official splits.

Benchmark	N	DBs	Dialect	Task	Composition	Metric
BIRD-Dev	1,534	11	SQLite	generation	by difficulty: simple 922, moderate 460, challenging 152	EX
BIRD-Critic	500	15	SQLite	debugging	by category: query 284, personalization 141, management 75	EX
LiveSQLBench	600	22	PostgreSQL	generation	by category: query 410, modification 190 by difficulty: high-level 286, non-high-level 314	EX
BIRD-Interact	600	22	PostgreSQL	conversational	by category: query 410, management 190 by difficulty: high-level 286, low-level 314	SR
Spider2-Lite	342	88	SQLite, Snowflake	generation	by dialect: SQLite 135, Snowflake 207	EX
Spider2-Snow	547	152	Snowflake	generation	—	EX
Spider2-DBT	64	64	DuckDB	project completion	—	DM

Table 4: The seven benchmarks: instance counts, databases, dialects, tasks, composition, and primary metrics (EX: execution accuracy; SR: task success rate; DM: database match; defined in Appendix A.2). The composition categories are defined in the per-benchmark descriptions of Appendix A.1.

BIRD-Dev (Li et al., 2023). The development split of BIRD: 1,534 natural language questions over 11 SQLite databases spanning 37 professional domains, each with optional external knowledge evidence. Difficulty tiers are assigned by the benchmark authors and reflect the complexity of the required SQL. We use the development split because the test split is hidden; as noted in Section 4.1, reported figures on this split vary by a small margin across works (Wretblad et al., 2024).

BIRD-Critic (Li et al., 2025b). The BIRD-Critic-SQLite release: 500 user issues, each consisting of a problem statement and (usually) a buggy SQL fragment that the agent must diagnose and fix. Query issues debug a failing analytical query; personalization issues tailor a query to a user-specific requirement beyond the literal bug; management issues repair statements that modify the schema or data. Instances are graded by benchmark-shipped test cases against the corrected query or statement.

LiveSQLBench (BIRD-bench Team, 2025). The LiveSQLBench-Base-Full v1 release: 600 instances over 22 full-scale PostgreSQL databases with hierarchical external knowledge documents. Query instances require analytical SELECTs; modification instances require DDL or DML graded by test cases. High-level instances pose the question through knowledge-base concepts, often composite metrics whose definitions the agent must resolve, while non-high-level instances state the requested computation directly.

BIRD-Interact (Huo et al., 2026). The full split of BIRD-Interact: 600 two-phase conversational instances over the same 22 PostgreSQL databases. The user is played by an LLM simulator that resolves ambiguities the benchmark deliberately injects; the agent acts through an ASK/SUBMIT protocol with a per-phase turn budget. An instance succeeds only if both phases pass. Query instances request analytical SELECTs and management instances request schema or data changes; the high-level and low-level split mirrors LiveSQLBench's, separating questions posed through knowledge-base concepts from directly stated ones. Phase-2 follow-ups build on the accepted Phase-1 answer and span five types: aggregation (120) summarizes the Phase-1 result; attribute change (114) adds, removes, or replaces reported columns; constraint change (49) alters the conditions of Phase 1; result-based follow-ups (266) pose a new question

that depends on the values Phase 1 returned; and topic pivot (51) shifts to a related question on the same database.

Spider2 (Lei et al., 2025). The Spider2 family targets enterprise-scale warehouses. Spider2-Lite and Spider2-Snow are single-query generation tasks against large schemas; Spider2-DBT asks the agent to complete a dbt project so that the final built database matches gold.

A.2 Metrics

For instance i of a benchmark with N instances, following the notation of Section 3, let y_i be the SQL the agent submits and $R_i = \text{exec}(y_i, \beta_i)$ its executed result on the instance database β_i; let y_i^* and R_i^* denote the benchmark's gold query and its result.

Execution accuracy (Li et al., 2023). The fraction of instances whose executed result matches gold,

$$\text{EX} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[R_i = R_i^*] \,,$$

where equality is taken under each benchmark's official comparison rules: row order only where the question requires it, per-instance decimal precision, and set semantics otherwise; where a benchmark provides several acceptable gold results, matching any one suffices. BIRD-Critic and LiveSQLBench replace the direct equality with benchmark-shipped test cases, so the indicator is one exactly when every test case passes against R_i or against the post-modification database state.

Soft-F1 (Li et al., 2023). Rows of R_i and R_i^* are aligned in order, and each aligned pair contributes the fraction of its cells that match to TP_i, with the unmatched fractions counted toward FP_i (prediction-only) and FN_i (gold-only); rows without a counterpart count wholly toward FP_i or FN_i. With per-instance precision $P_i = \text{TP}_i/(\text{TP}_i + \text{FP}_i)$ and recall $C_i = \text{TP}_i/(\text{TP}_i + \text{FN}_i)$:

$$\text{soft-F1} = \frac{1}{N} \sum_{i=1}^{N} \frac{2\, P_i\, C_i}{P_i + C_i}.$$

The metric grants partial credit for partially correct rows that execution accuracy scores as outright failures.

Valid Efficiency Score (Li et al., 2023). For each correctly answered instance, the gold and predicted execution times give a ratio $\tau_i = E(y_i^*)/E(y_i)$, which the benchmark's evaluation code maps to a step reward $\rho(\tau)$: 1.25 for $\tau \geq 2$, 1 for $\tau \in [1, 2)$, 0.75 for $\tau \in [0.5, 1)$, 0.5 for $\tau \in [0.25, 0.5)$, and 0.25 below:

$$\text{VES} = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[R_i = R_i^*] \cdot \sqrt{\rho(\tau_i)},$$

so a correct query at least as fast as gold earns full or bonus credit, and an incorrect one earns none.

Task success rate (Huo et al., 2026). Let $s_i^{(1)}, s_i^{(2)} \in \{0, 1\}$ indicate that phase 1 and phase 2 of instance i pass within their turn budgets. An instance succeeds only when both do:

$$\text{SR} = \frac{1}{N} \sum_{i=1}^{N} s_i^{(1)}\, s_i^{(2)}.$$

Normalized reward (Huo et al., 2026). The benchmark's phase-weighted score, which grants partial credit for a correct first phase using the official phase weights:

$$\text{reward} = \frac{1}{N} \sum_{i=1}^{N} \left(0.7\, s_i^{(1)} + 0.3\, s_i^{(2)} \right),$$

reported in percentage points.

Database match (Lei et al., 2025). For Spider2-DBT, building the agent's completed dbt project produces a database β'_i, whose benchmark-specified tables and columns are compared with the gold build β'^*_i:

$$\mathrm{DM} = \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}\left[\beta'_i = \beta'^*_i\right].$$

B Additional Results

The main paper reports a single headline score per benchmark (Table 1). This appendix reports the finer slices: category and phase breakdowns, difficulty and dialect breakdowns, and per-database results.

B.1 Category and phase breakdowns

Table 5 consolidates the sub-scores of the three benchmarks that ship finer task categories. Two patterns recur. Structured task variants score above their pure-query counterparts (BIRD-Critic management 78.7 against query 64.4, LiveSQLBench modification 66.3 against query 43.4), because modification tasks name their target objects and admit direct validation by execution. High-level questions cost fifteen to seventeen points on both PostgreSQL benchmarks (41.6 against 58.9 on LiveSQLBench, 47.6 against 63.1 on BIRD-Interact): resolving a knowledge-base concept into the right formula is harder than implementing a stated computation. On BIRD-Interact, phase 1 is the bottleneck (64.2 pass rate), while the conditional phase-2 rate of 86.8 shows that follow-ups rarely fail once a correct base answer exists.

Table 6 decomposes BIRD-Interact's second phase by follow-up type. Mechanical transformations of the accepted answer are the easiest: aggregation follow-ups pass 95.9% of the time conditioned on phase 1. Result-based follow-ups, which depend on the values the first answer returned, have the lowest conditional rate (82.4), and topic pivots score highest unconditionally (68.6) because they behave like fresh questions on a database the agent has already explored.

Slice	Total	Correct	Acc.
BIRD-Critic (by category)			
Query	284	183	64.4
Personalization	141	79	56.0
Management	75	59	78.7
LiveSQLBench (by category and difficulty)			
Query	410	178	43.4
Modification	190	126	66.3
High-level	286	119	41.6
Non-high-level	314	185	58.9
BIRD-Interact (by phase, category, difficulty)			
Phase-1 pass	600	385	64.2
Phase-2 (conditional)	385	334	86.8
Query	410	240	58.5
Management	190	94	49.5
High-level	286	136	47.6
Low-level	314	198	63.1

Table 5: Consolidated per-category, per-phase, and per-difficulty breakdown for the three benchmarks that report sub-scores. BIRD-Interact also reports a normalized reward of 61.6.

Follow-up type	N	Passed	Success (%)	Cond. (%)
aggregation	120	70	58.3	95.9
attribute change	114	72	63.2	86.7
constraint change	49	21	42.9	87.5
result-based	266	136	51.1	82.4
topic pivot	51	35	68.6	87.5

Table 6: BIRD-Interact results by phase-2 follow-up type (the types are defined in Appendix A.1). Cond. is the pass rate among instances whose Phase 1 passed; it is high in part because the runner echoes the accepted Phase-1 SQL into the Phase-2 prompt, anchoring the follow-up on the accepted base rather than a rewrite (see Appendix D).

B.2 Difficulty and dialect breakdowns

Table 7 reports BIRD-Dev by difficulty together with its additional official metrics. All three metrics fall monotonically with difficulty, and soft-F1 sits above execution accuracy at every tier with a gap that widens from 1.6 points on simple questions to 3.1 on challenging ones: harder questions are increasingly answered almost correctly, with partial row overlap that the strict metric scores as failure.

Table 8 splits Spider2-Lite by dialect. The SQLite subset scores seven points above the Snowflake subset (75.6 against 68.6), reflecting the larger schemas and semi-structured columns of the warehouse side; the gap is modest, consistent with the dialect robustness the main results show across benchmarks.

BIRD-Dev	**EX**	**soft-F1**	**VES**
Simple	83.2	84.8	77.4
Moderate	72.2	74.2	65.8
Challenging	61.2	64.3	56.6
Overall	77.7	79.2	71.9

Table 7: BIRD-Dev (1,534 instances) by difficulty, with the benchmark's additional official metrics (soft-F1, VES). As noted in the main paper, reported BIRD-Dev execution accuracy varies by a small margin across published works.

Dialect	N	Correct	EX (%)
SQLite	135	102	75.6
Snowflake	207	142	68.6
Total	**342**	**244**	**71.3**

Table 8: Spider2-Lite results by dialect.

B.3 Per-database results

Tables 9, 10, 11, and 12 report per-database results for LiveSQLBench, BIRD-Critic, BIRD-Interact, and BIRD-Dev, each sorted by accuracy. The spread is widest on LiveSQLBench, where per-database accuracy ranges from 76.5 to 19.2: the top of the table is well-specified operational data with clean joins, and the bottom is dominated by domain ambiguity in the knowledge base. LiveSQLBench and BIRD-Interact share the same 22 databases, and their rankings largely agree (`reverse_logistics` and `cybermarket_pattern` anchor the top of both tables while `mental_health` and `organ_transplant` anchor the bottom), indicating that difficulty is chiefly a property of the database and its knowledge base rather than of the interaction protocol. BIRD-Critic is more uniform: among its databases with at least ten issues, accuracy stays within a band from 54.8 to 75.0, suggesting debugging difficulty depends less on the domain than question answering does.

For the Spider2 family we report the distribution of per-database outcomes instead (Table 13): its databases hold only a few questions each, so individual per-database accuracies are coarse, and full tables over its 88, 152, and 64 databases would add pages without adding signal. Across Spider2-Lite and Spider2-Snow, DIA fully solves 99 of the 240 databases and is shut out on 40, so the headline accuracy

reflects broad competence rather than a few concentrated wins. Spider2-DBT is the hardest member, with 40 of 64 projects incomplete, consistent with project completion being the hardest task in Table 1. Per-database and per-project outcomes are part of the released per-instance results.

Database	Total	Q	M	All %
robot_fault_prediction	17	7/10	6/7	76.5
reverse_logistics	28	13/20	8/8	75.0
cybermarket_pattern	30	12/20	9/10	70.0
solar_panel	30	13/20	8/10	70.0
crypto_exchange	30	12/20	8/10	66.7
sports_events	29	11/20	8/9	65.5
exchange_traded_funds	28	13/19	4/9	60.7
fake_account	30	12/24	6/6	60.0
hulushows	30	10/20	8/10	60.0
museum_artifact	30	7/20	9/10	53.3
virtual_idol	28	7/19	7/9	50.0
households	29	6/21	8/8	48.3
polar_equipment	30	7/20	7/10	46.7
planets_data	29	10/19	3/10	44.8
cross_border	29	7/19	5/10	41.4
insider_trading	30	9/21	3/9	40.0
archeology_scan	13	3/10	2/3	38.5
disaster_relief	15	4/12	1/3	33.3
cold_chain_pharma_compliance	28	5/18	4/10	32.1
labor_certification_applications	29	6/19	3/10	31.0
organ_transplant	32	2/19	6/13	25.0
mental_health	26	2/20	3/6	19.2
Total	**600**	**178/410**	**126/190**	**50.7**

Table 9: LiveSQLBench per-database accuracy on the 600-instance complete run. Columns: total instances, Query category (correct/total), Modification category (correct/total), overall accuracy. Seven databases score above 60% and six below 40%; the bottom tier is dominated by domain ambiguity in the user knowledge base (`mental_health`, `organ_transplant`, `labor_certification_applications`), while the top tier is well-specified operational data with clean joins (`robot_fault_prediction`, `reverse_logistics`, `cybermarket_pattern`).

Database	Total	Correct	Accuracy (%)
esophageal	8	7	87.5
california_schools	24	18	75.0
toxicology	64	46	71.9
card_games	51	35	68.6
financial	44	30	68.2
debit_card_specializing	37	25	67.6
global_atlas	3	2	66.7
thrombosis_prediction	32	20	62.5
formula_1	44	27	61.4
superhero	40	24	60.0
codebase_community	12	7	58.3
european_football_2	69	40	58.0
erolp	28	16	57.1
student_club	42	23	54.8
spotify	2	1	50.0
Total	**500**	**321**	**64.2**

Table 10: BIRD-Critic per-database accuracy on the 500-instance complete run. The Management category is the cleanest tier: the agent benefits substantially from declaring the expected result shape before generation and from the identifier extractor that surfaces backtick-quoted names verbatim from the question text. Personalization is the hardest because gold often goes beyond what the question explicitly enumerates (e.g. adds derived columns, returns nested JSON instead of a flat result set), and the agent has no signal to predict gold's exact shape without seeing the test code.

Database	Total	Correct	All %
cybermarket_pattern	30	26	86.7
reverse_logistics	28	24	85.7
fake_account	30	25	83.3
insider_trading	30	21	70.0
crypto_exchange	30	20	66.7
solar_panel	30	20	66.7
exchange_traded_funds	28	17	60.7
disaster_relief	15	9	60.0
planets_data	29	17	58.6
households	29	16	55.2
sports_events	29	16	55.2
hulushows	30	16	53.3
virtual_idol	28	14	50.0
labor_certification_applications	29	14	48.3
robot_fault_prediction	17	8	47.1
polar_equipment	30	14	46.7
cold_chain_pharma_compliance	28	13	46.4
archeology_scan	13	5	38.5
cross_border	29	11	37.9
organ_transplant	32	11	34.4
museum_artifact	30	10	33.3
mental_health	26	7	26.9
Total	**600**	**334**	**55.7**

Table 11: BIRD-Interact per-database success rate on the 600-instance run. Phase, category, and difficulty aggregates are in Table 5; results by follow-up type are in Table 6.

Database	N	Correct	EX	soft-F1	VES
superhero	129	117	90.7	92.9	83.9
student_club	158	143	90.5	92.2	84.5
debit_card_specializing	64	54	84.4	88.4	78.3
codebase_community	186	153	82.3	85.0	76.8
european_football_2	129	101	78.3	81.0	72.8
toxicology	145	111	76.6	75.7	70.4
card_games	191	145	75.9	75.8	70.8
formula_1	174	130	74.7	74.3	68.8
california_schools	89	64	71.9	77.5	66.4
thrombosis_prediction	163	107	65.6	67.4	58.4
financial	106	67	63.2	69.2	59.0
Total	**1,534**	**1,192**	**77.7**	**79.2**	**71.9**

Table 12: BIRD-Dev per-database results, sorted by execution accuracy.

Benchmark	DBs	Fully solved	Partially solved	Unsolved
Spider2-Lite	88	36	37	15
Spider2-Snow	152	63	64	25
Spider2-DBT	64	24	—	40

Table 13: Distribution of per-database outcomes on the Spider2 family: databases where every question is answered correctly, where some are, and where none are. Spider2-DBT instances are single whole-project completions, so no partial bucket applies.

C Error Analysis

This appendix expands the error analysis of Section 4.6. The finding that organizes it is that the remaining headroom is semantic rather than syntactic: on every benchmark almost every failed instance executes cleanly and returns a wrong answer. Almost all of them are semantic, and we sort them into three recurring classes, each pointing to a different remedy. Reasoning failures, where the query answers a subtly different

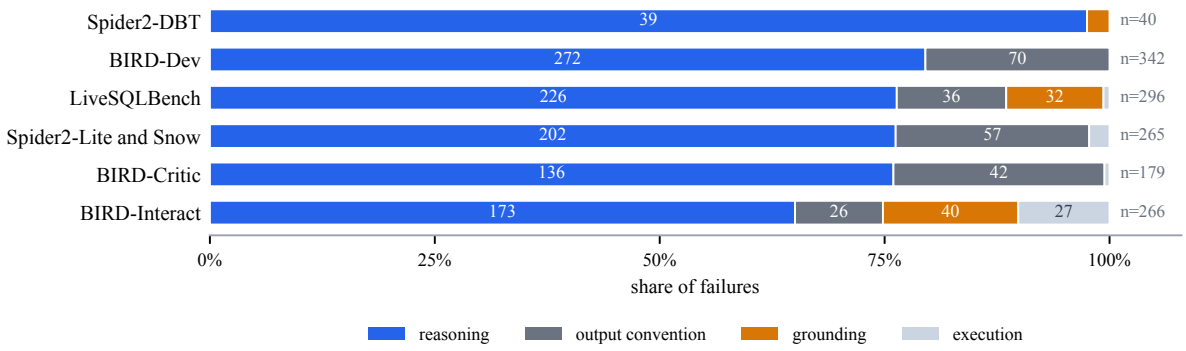

Figure 3: Composition of failures per benchmark, aggregated over each benchmark's task categories and ordered by the share of reasoning failures. Segment labels are failure counts.

question than the one asked, are the largest class on every benchmark and are bound to model capability. Output-convention failures, where the right quantity is computed but presented in the wrong shape, are the most addressable class and respond to refinements of the standing instructions. Grounding failures, where the agent cannot resolve an exact identifier, appear on the benchmarks with object-modification tasks and recur within a database, the recurrence structure that memory targets (Section 4.5). Execution failures, where the SQL does not run at all, are rare, under three percent of failures in aggregate.

We classify every failure of each benchmark's reported run by comparing the agent's output to gold, directly from the per-instance results: 342 failures on BIRD-Dev, 296 on LiveSQLBench, 266 on BIRD-Interact, 179 on BIRD-Critic, 98 on Spider2-Lite, 167 on Spider2-Snow, and 40 on Spider2-DBT. Where the results record executed result tables, on BIRD-Dev, LiveSQLBench query tasks, and the Spider2 family, we compare predicted and gold result shapes and values; where they record only pass or fail against hidden tests, on BIRD-Critic and the modification tasks, we compare the predicted and gold SQL. Two limits follow. Output convention is detected only where the result shape is observable, so on the SQL-only failures its share is a lower bound; and a wrong literal value, which is a grounding error in spirit, is indistinguishable from wrong logic once it reaches the result, so on the pure query benchmarks it falls under reasoning.

C.1 Failure classes across benchmarks

Figure 3 reports the class composition of failures per benchmark.

Reasoning failures dominate every benchmark, from roughly three quarters on most to two thirds of BIRD-Interact's. The class covers three recurring shapes. Selection errors choose the wrong rows through a wrong filter, join path, or entity, and account for the largest buckets of Table 14. Computation errors feed the right rows into a wrong formula or aggregate at the wrong granularity, and drive the family-wide signature of same-shape results with wrong values. Construct errors misuse a SQL idiom, such as an inner join where the question implies an outer one, ties dropped at a top-N boundary, or a defensive NULL filter that removes valid rows, and are among the structurally identifiable patterns of Table 15. All three execute cleanly and concentrate where the question under-specifies its intent; none respond to instruction refinements, marking the capability frontier of the model rather than a fixable gap in the system.

Output-convention failures are the largest addressable class. Extra or missing output columns are the main form, with row order making up the rest. The computed quantity is right and only its presentation diverges from what the gold answer admits, which is why these patterns respond to standing-instruction refinements where reasoning failures do not. Because convention is counted only where the result shape is observable, its share on the SQL-only benchmarks understates the true total.

Grounding failures concentrate on the benchmarks with object-modification tasks. Most of LiveSQL-Bench's and BIRD-Interact's grounding failures trace to table, view, or column names the agent paraphrased, so a probe written against the gold name cannot find them. These recur within a database, the same identifiers returning question after question, which is the recurrence structure that memory targets

(Section 4.5).

C.2 Per-benchmark patterns

Tables 14 and 15 give the per-pattern breakdown for the two benchmarks whose results expose the most detail: LiveSQLBench, which records executed result tables for its query tasks, and BIRD-Critic, whose pass or fail outcomes we read through the predicted and gold SQL.

Pattern	Count	Diagnosis
wrong rows returned	105	wrong filter, join path, or entity: the returned rows barely overlap gold
row count moderately off	58	wrong filter or join cardinality, within a factor of five
row count far off	22	major logic error, off by more than a factor of five
extra output columns	16	display columns the gold answer does not admit
missing output columns	14	projection narrowed past the requested columns
right rows, wrong values	7	correct rows but wrong computation or formula
wrong row order	6	correct rows in the wrong order
empty result	4	the query returned no rows

Table 14: LiveSQLBench query failures (232), by comparison of the executed result with gold. The 64 modification failures are test-case-graded and split into grounding (32, paraphrased identifiers), reasoning (30, logic errors), and execution (2).

Pattern	Count	Diagnosis
extra output columns	23	output decorated with the question's free-text dimensions
missing output columns	19	projection narrowed past the columns the user named
unrequested NULL filter	23	defensive `IS NOT NULL` that drops valid rows
dropped ties	8	`ROW_NUMBER() = 1` or `LIMIT 1` where gold keeps ties
overcomplicated SQL	5	recursive CTEs or manual date arithmetic where a simpler idiom suffices
wrong join type	2	`INNER` where the question implies `LEFT`, or the reverse

Table 15: BIRD-Critic failures with a structurally identifiable pattern in the SQL (80 of 179). The remaining 98 are value or logic errors not separable from the SQL text alone, and one is an execution error. Extra or missing output columns, the convention class, are the largest identifiable patterns and recur across the query, personalization, and management categories alike.

For the Spider2 family the same result-shape comparison applies directly. Of Spider2-Lite and Spider2-Snow's 265 query failures, 202 return a result of gold's shape with wrong values or row counts, 57 differ in column count, and 6 fail to execute. The 40 Spider2-DBT project failures, taken at the first mismatching table of the built database, are almost all content errors: 39 build the required tables but with wrong contents, and 1 leaves a required table uncreated so a downstream reference errors. Because dbt grades by whole-table equality with no result tuples to inspect, column-count divergences cannot be separated from content errors here, so the convention share folds into the first bucket and the lower-bound caveat above applies most strongly to this benchmark. Across the family wrong values and row counts dominate, extra or missing columns are the main addressable slice, and outright execution errors stay near two percent, consistent with Section 4.6.

C.3 Interaction failure behaviour

BIRD-Interact adds a behavioural failure dimension that the single-shot benchmarks lack. Its 266 failures split into 215 in phase 1 and 51 in phase 2.

The dominant behaviour is hypothesis lock-in: 215 of the 266 failures (81%) are phase-1 trajectories that exhausted the 15-turn budget, the agent re-submitting near-identical SQL against the protocol's minimal feedback instead of stepping back to ask a clarifying question. The recurring modes are inferring a composite formula without asking, chasing sort order when the real divergence was a formula input, and phase-1 views whose paraphrased column names break the gold phase-2 SQL. These modes map onto the

same classes as the single-shot benchmarks: composite-formula inference is a reasoning failure, sort-order chasing an output-convention failure, and paraphrased identifiers a grounding failure. Classifying the final submissions accordingly gives the BIRD-Interact bar of Figure 3: 173 reasoning, 40 grounding from paraphrased identifiers on management tasks, 26 output-convention, and 27 execution errors from malformed submissions at the turn cap. Worked traces of the recurring modes appear in Appendix D, together with the interaction policies they motivated.

D Case Studies

The premise of this work is that an agent grounded in execution can answer reliably because it can verify its work against the database. BIRD-Interact is the sharpest test of that premise, and the benchmark where DIA's margin over prior work is largest (Table 1), because it is the one setting where part of the ground truth does not live in the database: the user's intended formula and the canonical names of requested artifacts are known only to the user. Execution grounding answers everything the data can answer; dialogue must cover the rest, and knowing the boundary between the two is the skill the benchmark rewards. It is also the benchmark that most resembles our production deployment, where domain experts pose under-specified questions conversationally and clarification is part of normal operation (Appendix G). This appendix shows that boundary in worked traces: one released-run pass where grounding and a single targeted question divide the work correctly, and two failure modes, a stuck-loop and a Phase-2 cascade, that we observed during development and that motivated the interaction policies described at the end. We pair each failure trace with the released-run outcome of that same instance once the policies are in place.

D.1 The BIRD-Interact protocol

Each BIRD-Interact instance is a multi-turn conversation between the *Query Generator* and the benchmark's LLM-driven user simulator. Three properties make it qualitatively different from the other six benchmarks in our evaluation set:

- Two phases per instance. Phase 1 is the primary question (`Q` or `M`); Phase 2 is a follow-up that builds on the agent's Phase 1 answer. The agent must SUBMIT a SQL for Phase 1; if it passes, Phase 2 issues a new prompt that references Phase 1's result shape (e.g. "filter the result you just produced to rows where . . .").
- An ASK or SUBMIT protocol with a fixed turn cap per phase (Appendix I). Every turn the agent either emits `ASK: <question>` to request clarification or `SUBMIT: <sql>` to attempt an answer. ASKs are routed to the user simulator (an LLM playing the role of a non-SQL domain expert); SUBMITs are evaluated, and an INCORRECT verdict returns a structural delta (row count, column count) without disclosing gold values. This is the benchmark's information-asymmetric protocol, matching what real users could actually provide.
- Follow-ups bind to Phase-1 artifacts. For Management tasks, Phase 2 references the object the agent created in Phase 1, including its column names. This surfaces a cascade failure mode unique to BIRD-Interact: if Phase 1 paraphrased a column name, the Phase 2 evaluation cannot see it (see the Phase-2 cascade trace).

The combination produces a benchmark where 81% of failures burn the full turn cap on a single phase (the agent keeps re-submitting near-identical SQL against the same minimal feedback), and where the decisive skill is not which SQL to emit but when to stop guessing and ask.

Figure 4 gives a single-run view of what the benchmark's authors term interaction-time scaling (Huo et al., 2026), the ability to convert multi-turn interaction into end performance: rather than re-running the benchmark at each budget, it reports the turn at which each passing submission landed under the full budget. The curve is cumulative, so what is informative is its shape: DIA exceeds the final score of the best prior system within its first four turns, is still climbing steeply late in the budget (turns eleven through fifteen add another nine points), and flattens only past twenty total turns, so the cap leaves little on the table. We do not evaluate the benchmark's stronger criterion (the interaction-time scaling law),

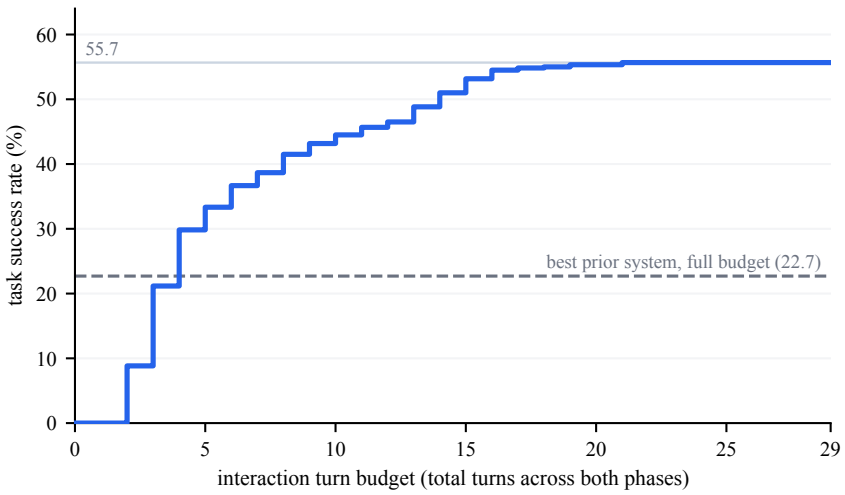

Figure 4: Interaction-time scaling on BIRD-Interact: the fraction of the 600 instances whose passing submission landed within the first k total turns across both phases, computed from the single full-budget run. The dashed line is the final score of the best prior system at its full budget.

which compares interactive performance against idealized single-turn performance on the disambiguated task. Successful instances are efficient, finishing in a median of four turns, while failed instances are the ones pinned at the cap. Under-specification also has a measurable dialogue cost even in success: passing high-level instances need a median of six turns against four for low-level ones, the price of resolving a composite metric through conversation.

D.2 A passing trace

The passing trace shows the division of labour working: one targeted ASK for the formula that execution cannot reveal, workspace probes for everything it can, and a first-submission pass on both phases.

Passing trace (archeology_scan instance 1)

```
TURN 1
USER: For each scan site, report the average ambient temperature,
humidity, and illumination, along with the Environmental
Sensitivity Index (ESI). Order by ESI descending.

AGENT: ASK: The KB defines ESI in terms of three sub-metrics
(ambient_temp_norm, humidity_norm, illumination_norm). Could you
confirm whether ESI is the simple mean of the three normalized
values, or a weighted combination?

USER_SIM: labeled("ESI"). Each row of the scans table has an
ambient_cond JSON object with Ambic_Temp, Ambic_Humidity, and
Ambic_Light keys. ESI is the simple mean of the three normalized
values, after each value is min-max normalized over the entire
table. Use COALESCE(<expr>::real, <default>) with the defaults
listed in the KB entry for ESI when a key is missing.

TURN 2
AGENT (workspace): writes and runs a probe script before
submitting:

  keys = query_db("SELECT DISTINCT jsonb_object_keys(ambient_cond)
                   FROM scans")
  # -> Ambic_Temp, Ambic_Humidity, Ambic_Light

  missing = query_db("SELECT COUNT(*) FROM scans
                      WHERE NOT ambient_cond ? 'Ambic_Temp'")
  # -> some rows lack the key: the COALESCE defaults matter
```

```
AGENT: SUBMIT:
WITH normed AS (
  SELECT s.arcref AS site_id,
         COALESCE((s.ambient_cond->>'Ambic_Temp')::real, 20.0) AS t,
         COALESCE((s.ambient_cond->>'Ambic_Humidity')::real, 50.0) AS h,
         COALESCE((s.ambient_cond->>'Ambic_Light')::real, 300.0) AS l
  FROM scans s
),
bounds AS (
  SELECT MIN(t) AS t_min, MAX(t) AS t_max,
         MIN(h) AS h_min, MAX(h) AS h_max,
         MIN(l) AS l_min, MAX(l) AS l_max
  FROM normed
)
SELECT n.site_id,
       AVG(n.t) AS avg_temp,
       AVG(n.h) AS avg_humidity,
       AVG(n.l) AS avg_illumination,
       AVG( (n.t - b.t_min) / NULLIF(b.t_max - b.t_min, 0)
          + (n.h - b.h_min) / NULLIF(b.h_max - b.h_min, 0)
          + (n.l - b.l_min) / NULLIF(b.l_max - b.l_min, 0) ) / 3.0 AS esi
FROM normed n CROSS JOIN bounds b
GROUP BY n.site_id
ORDER BY esi DESC;

EVALUATOR: PASS (rows=147, columns=5, match within 1 ULP)

PHASE 2 prompt: Restrict the result to the top 10 most-sensitive
sites and add a classification column ('high'/'medium'/'low')
based on NTILE(3) over ESI.

AGENT: SUBMIT:
WITH p1 AS (
  -- accepted phase-1 SQL, carried forward as the base
  ... [as above] ...
)
SELECT site_id, avg_temp, avg_humidity, avg_illumination, esi,
       CASE NTILE(3) OVER (ORDER BY esi DESC)
         WHEN 1 THEN 'high' WHEN 2 THEN 'medium' ELSE 'low' END
         AS sensitivity_class
FROM p1
ORDER BY esi DESC
LIMIT 10;

EVALUATOR: PASS. Final verdict: PASS (both phases).
```

Three properties of this trace are diagnostic of when the agent succeeds: (i) it divides the work correctly between grounding and dialogue, probing the database for what execution can answer (key spellings, missing-key rates) and ASKing only for what it cannot (the formula definition); (ii) it uses the KB-listed COALESCE defaults verbatim (the stuck-loop trace shows the failure mode this avoids); and (iii) Phase 2 wraps the accepted Phase 1 SQL in a CTE rather than rewriting from scratch (DIA carries the accepted Phase-1 SQL into the follow-up context, making this the natural default).

D.3 A stuck-loop failure

This trace, recorded during development before the clarification policy below was in place, shows what was then the most common BIRD-Interact failure mode: the agent infers a composite formula from KB names without asking, then burns the entire turn budget with cosmetic edits (cast types, JSON-key spellings) instead of stepping back to ASK.

Stuck-loop trace (archeology_scan instance 5, development run): turn budget exhausted with 0 ASKs

```
KB CONTEXT: DPQ = 0.4*SQS + 0.4*MFS + 0.3*SCE_mod + 0.2*RAR,
where each sub-term is itself a multi-term expression over
JSON keys masked from the agent.

TURN 1
USER: For each scan location, compute a Data Provenance Quality
index (DPQ).

AGENT: SUBMIT: -- guessed composite from KB term names
SELECT zoneref,
       0.4*sqs + 0.4*mfs + 0.3*sce + 0.2*rar AS dpq
FROM (
  SELECT zoneref,
         AVG((meta->>'survey_quality_score')::real)    AS sqs,
         AVG((meta->>'metadata_fidelity_score')::real) AS mfs,
         AVG((meta->>'site_completeness_eval')::real)  AS sce,
         AVG((meta->>'reference_alignment')::real)     AS rar
  FROM scans GROUP BY zoneref
) t
ORDER BY dpq DESC;

EVALUATOR: INCORRECT. (Row and column counts match gold;
  values differ. No gold values disclosed.)

TURN 2 ... TURN 14: agent re-SUBMITs 14 near-identical variants:
swaps `::real` for `::numeric`, varies the JSON keys
(`survey_quality_score` -> `sqs_raw` -> `surveyqualscore`),
wraps in additional CTEs, adds ORDER BY ties. Every SUBMIT
returns the same verdict, with no gold values disclosed.

TURN 15 (final turn)
AGENT: SUBMIT: -- final cosmetic variant, still wrong
... [as before, with one cast rewritten] ...

EVALUATOR: INCORRECT. Final verdict: FAIL (Phase-1 not passed,
Phase-2 not attempted).
```

The agent never ASKs to clarify either (a) what the masked sub-terms actually compute (the KB lists their names but not the formulas they expand to) or (b) whether the composite is computed per-row before aggregation or after. Both are required to reach gold's roughly 5,000-character composite. This is a boundary failure of execution grounding: the formula's semantics live with the user, not in the data, so no amount of probing or re-execution can verify a guessed composite. The agent's mistake is treating a dialogue problem as a grounding problem. Standing guidance alone does not prevent this mode: nothing behavioural forces the agent to back off from a losing SUBMIT to an ASK, which is what motivated the clarification-question policy. With that policy active in the released run, pure stuck loops are nearly eliminated: only two of the 215 Phase-1 failures issue as few as one clarifying question and none issue zero, and this same instance now issues several ASKs before exhausting its budget. The trace shows the failure mode the policy targets, not residual released behaviour.

D.4 A Phase-2 cascade

This trace, also from the development run, shows a class of failure unique to BIRD-Interact Management: the agent's Phase-1 view shape is locally correct but missing a column that the Phase-2 evaluation will then need to reference.

Phase-2 cascade trace (archeology_scan M_3, development run)

```
PHASE 1
USER: Create a view `view_premium_quality_scans` containing scan
ID, project, site, timestamp, resolution, density, coverage,
overlap, and noise, restricted to scans whose DPQ is above the
```

```
75th percentile.

AGENT: SUBMIT:
CREATE OR REPLACE VIEW view_premium_quality_scans AS
WITH q AS (... DPQ computation ...)
SELECT s.facetregistry AS scan_id,
       s.project,
       s.facetregistry AS site_code,    -- paraphrased "site"
       s.captured_at  AS timestamp,
       s.resolution_m AS resolution,
       ...
FROM scans s JOIN q USING (zoneref)
WHERE q.dpq > q.dpq_p75;

EVALUATOR: PASS (Phase 1 view created; rows match gold count).

PHASE 2
USER: For each project in `view_premium_quality_scans`, report
the count of premium scans and the average noise level.

AGENT: SUBMIT:
SELECT project, COUNT(*) AS n, AVG(noise) AS avg_noise
FROM view_premium_quality_scans
GROUP BY project;

EVALUATOR: the Phase-2 evaluation references the view's
underlying column names:
  SELECT zoneref, COUNT(*), AVG(noise)
  FROM view_premium_quality_scans GROUP BY zoneref;
... which raises UndefinedColumn: column
view_premium_quality_scans.zoneref does not exist.
(The agent aliased site to site_code.)

Verdict: FAIL (Phase-2 cascade due to Phase-1 column-name drift).
```

The agent's Phase-1 view is functionally correct: the row set matches gold and the projection covers the columns the question enumerated. It fails not because of its own Phase-2 SQL but because the follow-up's expected answer is keyed to the underlying column names: the Phase-2 evaluation references `zoneref` (the table's actual column name) while the agent paraphrased it to `site_code`. This is the other boundary failure of execution grounding: execution can validate that the view runs and its rows match, but it cannot reveal the canonical names a follow-up will expect, because naming is a matter of user intent rather than data. DIA's standing instructions therefore treat a Management projection list as not a renaming contract: when the question names a column conversationally ("site"), the agent projects the underlying column under its original identifier rather than inventing a conversational alias. With this policy in place, the same instance passes both phases in the released run: the Phase-1 view exposes `zoneref` under its own name, so the Phase-2 follow-up resolves.

D.5 Interaction policies

These failure modes shaped how the *Query Generator* behaves in multi-turn protocols. Four mechanisms carry most of the weight.

Forced clarification. After three consecutive INCORRECT verdicts with the same structural signature, DIA requires its next action to be an ASK rather than another SUBMIT. Stuck loops are the largest behavioural failure mode, and this policy cuts them directly. The streak resets on any ASK or on a SUBMIT whose verdict differs. Ablating this policy across the full 600-instance run drops accuracy from 55.7% to 14.5%, a 41.2-point fall, confirming that stuck loops, not a limit on the model's SQL ability, are what the policy is protecting against.

Resubmission control. DIA does not resubmit near-identical SQL: a candidate SUBMIT that matches one of its recent attempts is rejected, and the agent must either ASK or change the structural approach. This closes the repeated-resubmission pattern within stuck-loop traces.

Context carryover. The accepted Phase-1 SQL is placed at the head of the Phase-2 context, so the natural default is to apply the requested follow-up edit to the Phase-1 base, which is the protocol's intent, rather than rewriting from scratch and silently changing the Phase-1 filter or projection.

Short-term and long-term memory. Memory extends the same boundary, and in deployment we see this in practice. Within a conversation, clarifications act as short-term memory: once the user states a formula, every later turn builds on it. Across conversations, those answers become long-term memory: the same domain experts return with the same vocabulary, and a formula or naming convention they have already explained is not asked for again. Our treatment of memory (Section 4.5) concerns recurrence within a database rather than within a user.

Together these mechanisms encode the boundary the traces illustrate: the agent grounds everything the database can answer by execution and spends dialogue only on what it cannot. That division of labour, rather than any difference in SQL fluency, is where the margin on this benchmark comes from (Table 1).

E Standing Instructions

The standing instructions of the *Query Generator* share one architecture across benchmarks. Every seed file combines three ingredients: an output-contract discipline, under which the agent declares the expected shape of its answer and verifies the executed result against it; reference material and pitfalls for the SQL dialect; and guidance for the task format. The per-question prompt itself is thin: it lays out the workspace, states the question and its task metadata, and points the agent at the seed. Table 16 summarizes how each benchmark instantiates this architecture.

Benchmark	Task-specific content of its seed
BIRD-Dev	projection discipline; verification of literal values and transforms against the live database; memory usage
BIRD-Critic	a five-step debugging workflow; patterns for the query, personalization, and management categories
LiveSQLBench	the same workflow adapted to generation; data-modification guidance; external-knowledge usage rules
BIRD-Interact	conversational guidance: when to ask versus when to submit, what the simulated user can answer, and management-task fidelity
Spider2-Lite	a compact output contract and semantic rules
Spider2-Snow	the same contract instantiated for Snowflake
Spider2-DBT	a project-completion workflow: analyze the dbt project, build, and verify the produced database

Table 16: How each benchmark instantiates the shared architecture: the task-specific content of its seed, beyond the output-contract discipline and the dialect reference.

E.1 The workflow skeleton

The fullest form of the contract discipline is a five-step workflow used by the debugging, modification, and conversational benchmarks: plan with an explicit contract, diagnose by execution, build, validate against the contract, save. We reproduce a condensed BIRD-Critic instantiation.

Workflow skeleton as instantiated for BIRD-Critic (STEP 1 to STEP 5)

```
## Per-Question Workflow

### STEP 1 - PLAN
- Read the user's question and any buggy SQL carefully.
- Identify the category (Query / Personalization / Management).
- Define an explicit OUTPUT_CONTRACT:
    cols:    [c1, c2, ...]
    rows:    ~N (one per <entity>)
    order:   <unordered | by X asc/desc>
    filters: [every constraint expressed in the question]
- Filters checklist (CRITICAL): walk every adjective, prepositional
  phrase, "only/excluding/ignoring", and conjunction in the prose.
  Each becomes a row in `filters: [...]`.
```

```
### STEP 2 - DIAGNOSE
- Inspect the schema (PRAGMA table_info / information_schema).
- Execute the buggy SQL (if any) and observe its actual output.
- Compare actual output to OUTPUT_CONTRACT. What's wrong?

### STEP 3 - BUILD
- Projection: entity-key -> metric -> drop redundant display cols
  -> drop helper window cols from outer SELECT.
- Minimal-diff: smallest edit to buggy SQL that fixes the bug.
- Apply every filter in the STEP 1 checklist.

### STEP 4 - VALIDATE
- Run pred via query_db; observe the result set.
- For every item in `filters`, point at the clause that implements it.
- Check column count, row count plausibility, JOIN cardinality,
  first-row sanity, and absence of spurious NULL filters.

### STEP 5 - SAVE
- save_result(corrected_sql, result_rows, tables_used)
- For Management tasks save the DDL/DML statements themselves
  (not the verification SELECT).
- cleanup() to reset the workspace for the next question.
```

The contract is declared before any SQL is run, so STEP 4 can fail an answer on shape grounds using nothing but the agent's own plan. This is the self-verification loop of the main paper in operational form, and it targets the largest addressable failure class of Appendix C: extra display columns, missing filters, and helper columns leaking into the outer projection.

E.2 A worked debugging example

The trace below is an illustrative reconstruction, composed to show the five-step contract discipline end to end rather than transcribed from a single released instance. It walks the workflow through a representative BIRD-Critic debugging question over the `financial` database: the agent receives the user's prose and a buggy SQL fragment, diagnoses the bug by executing it, and repairs it with a minimal edit.

Worked trace (illustrative; BIRD-Critic, financial database)

```
QUESTION: For each account, what is the latest transaction date
where the amount exceeded the account's average transaction
amount? List account_id and the transaction date.

BUGGY SQL (issued by the user):
  SELECT a.account_id, MAX(t.date)
  FROM account a
  JOIN trans t ON a.account_id = t.account_id
  WHERE t.amount > AVG(t.amount)
  GROUP BY a.account_id;

STEP 1 PLAN: the agent declares its contract.
  OUTPUT_CONTRACT:
    cols:    [account_id, latest_date]
    rows:    ~one per account (~4500)
    order:   none specified
    filters: [transactions whose amount > account-level mean]

STEP 2 DIAGNOSE: the agent executes the buggy SQL and observes
  SQLite error: misuse of aggregate function AVG()
The aggregate sits inside a row-level WHERE, which is invalid:
the per-account mean must be computed per group.

STEP 3 BUILD: minimal repair with a window function.
  WITH t_with_avg AS (
    SELECT t.account_id, t.date, t.amount,
           AVG(t.amount) OVER (PARTITION BY t.account_id) AS acct_avg
```

```
    FROM trans t
  )
  SELECT account_id, MAX(date) AS latest_date
  FROM t_with_avg
  WHERE amount > acct_avg
  GROUP BY account_id;

STEP 4 VALIDATE: the agent runs the corrected SQL via query_db.
  4500 rows, 2 columns
Contract check: column count matches (2 = 2); one row per
account; the filter "amount above account-level mean" is
implemented by WHERE amount > acct_avg; no spurious
IS NOT NULL filter was added.

STEP 5 SAVE: save_result(corrected_sql, result_rows,
  tables_used=[trans, account]); cleanup().
```

The repair is the minimal edit the seed asks for: the structure of the buggy SQL is preserved, and only the invalid aggregate placement changes. The contract declared in STEP 1 is what makes STEP 4 a real check rather than a formality: every entry in it is verified against the executed result before the answer is saved.

F Memory: Store and Contents

This appendix gives concrete form to the memory introduced in Section 3.2: how the experience store is organized, how its entries are written, and what kinds of knowledge it captures during a run. The view is observational, drawn from the BIRD-Dev run.

F.1 The three tiers

Memory is held as files in the workspace and organized in three tiers.

Retrieved examples. The pool for this tier is the BIRD training split: its gold question-and-SQL pairs together with per-table column-meaning notes, embedded once offline into a fixed similarity index that is reused unchanged across runs. The split is disjoint from the BIRD-Dev evaluation set, so retrieval introduces no test leakage. For each question, the few most similar pairs above a fixed similarity threshold (Appendix I) are surfaced and staged for the agent to consult. These are concrete examples rather than distilled rules: the agent reads them as idioms and re-derives any abstraction at the point of use, confirming column names and values against the current database before relying on them.

Session lessons. While working on a database, the agent reflects after each question and records a short conditional rule when something it tried held, together with the observation that confirmed it. A consolidation step keeps a compact set per database rather than every candidate.

Cross-session lessons. The subset of session lessons that recur across databases, rather than holding only within one, is promoted to a persistent store and carried into later tasks. Promotion is outcome-gated: a rule is retained only when later questions continue to bear it out.

F.2 Representative rules

Across the BIRD-Dev databases the promoted rules fall into a few recurring kinds: aggregating without double-counting across one-to-many joins, choosing the join path that carries a given attribute, recognizing when a stored value is already a ratio rather than a percentage, projecting bridge tables without duplicate rows, and reading compound filter phrasing. Each is a short, human-readable conditional paired with the evidence that confirmed it. Table 17 gives representative rules with that evidence.

Because each rule is written by the agent itself and re-checked on the live database before it can change an answer (Section 3.2), the store stays interpretable and auditable: a domain expert can read any entry, see the evidence behind it, and judge whether it should apply, in the same spirit as the other artifacts DIA produces.

Kind	Rule	Evidence
Output format	Compute a percentage from counts; do not read a similarly named stored column, which often holds a 0–1 ratio.	A stored column held about 0.70 where the question wanted 70.15 computed.
Filter value	A strict "oldest and lowest X" intersection often returns nothing; probe it, then read it as "oldest within the lowest-X group."	The strict conjunction returned 0 rows; the relaxed reading returned the intended single row.
Filter value	Probe the stored casing of status or legality enums; many are Title Case, so a lowercase literal silently matches nothing.	Filters passed only as `'Banned'/'Restricted'`, not the lowercase values a hint used.
Projection	Querying IDs from a many-to-many bridge table can return an entity several times; add `DISTINCT` only when the question asks for unique entities.	Gold returned one entity ID four times, all at the same minimum value.
Aggregation	"Spent the most" means `SUM` over transactions, not `MAX` of a single one, even when a hint says `MAX`.	Ordering people by `MAX(cost)` failed; summed spending was the intended measure.
Output format	Return only the `YYYY-MM-DD` part of a datetime column when the question asks for a date.	A full timestamp mismatched the bare date the gold answer expected.
Filter value	"After year X" means $> X$, even when a hint maps it to $= X$.	`= '1990'` returned that year's records rather than later ones.

Table 17: Representative cross-session rules with the execution observation that confirmed each. Rules are condensed; each is recorded only after the observation holds on the database at hand.

F.3 Episodic-to-semantic generalization

A rule does not begin general. The agent first records a concrete observation on the database it is working, and that within-database lesson is promoted to a cross-session rule only when later questions bear out the same pattern. Table 18 shows this episodic-to-semantic step for two rules: the left column is the originating observation, the right is the generalized rule the store keeps.

Within-database observation	Cross-database rule
On `california_schools`, counting schools through the school-to-`frpm` join, `COUNT(*)` returned 9,977 for one school because it had 9,977 fact rows, while `COUNT(DISTINCT CDSCode)` returned 1.	When counting entities through a join to a table that stores several rows per entity, count with `COUNT(DISTINCT entity_pk)` rather than `COUNT(*)`.
On `financial`, filtering "account branch in East Bohemia" through the client's home district returned the wrong population; the account's own district was meant.	Join through the entity that owns the attribute the question names: the account's geography for "account branch," the client's only for "client lives in."

Table 18: The episodic-to-semantic step. A rule begins as a concrete observation on one database and is promoted to a cross-session rule only when it generalizes beyond that database.

F.4 A learned rule in use

The tiers are consulted, not only written. The trace below shows a session lesson, formed on earlier questions of `thrombosis_prediction` and re-checked by a live probe before use, redirecting a later answer from a per-patient collapse to the per-record projection the question intends. We include it to illustrate the mechanism, not as a measurement of memory's aggregate effect.

A session lesson redirects the output (thrombosis_prediction)

```
QUESTION: for each patient born in 1982, state whether their
albumin (ALB) is within the normal range, 3.5 to 5.5.

SESSION LESSON (formed on earlier questions of this database):
  [JOIN-PATH] The Patient->Laboratory join fans out to many lab
  records per patient, so a per-row label is over records, not
  patients. Probe the join size before choosing output granularity.
```

```
UNAIDED ATTEMPT: groups by patient, one label per patient.
  SELECT P.ID, CASE WHEN ... THEN 'normal' ELSE 'abnormal' END
  FROM Patient P LEFT JOIN Laboratory L ON P.ID = L.ID
  WHERE P.Birthday LIKE '1982%'
  GROUP BY P.ID;
  --> collapses the fan-out to one row per patient (wrong shape).

WITH THE LESSON IN CONTEXT: probe the join first.
  COUNT(DISTINCT P.ID) = 1     COUNT(*) = 35
  (one matching patient, thirty-five lab records)
Then label one row per record:
  SELECT IIF(L.ALB BETWEEN 3.5 AND 5.5, 'normal', 'abnormal')
  FROM Patient P JOIN Laboratory L ON P.ID = L.ID
  WHERE STRFTIME('%Y', P.Birthday) = '1982';
  --> 35 rows, one per lab record: matches gold.
```

G Production Deployment

DIA is deployed in production for enterprise customers. This appendix illustrates a deployment through one workflow, a nursing-staff analysis, carried out as a single conversation. The domain expert uploads a set of operational data files, and in one continuous thread the three agents work in turn over a shared workspace: each builds on the artifacts the previous one produced, and every artifact, the interpretation, the schema, and the analyses, is retained and remains visible for review. The figures are screenshots from this conversation.

Data Interpreter. The *Data Interpreter* inspects the uploaded files and recovers their structure: the entities, the relationships among them, and any data-quality issues. It presents these findings for the domain expert to confirm or correct rather than assuming them (Figure 5).

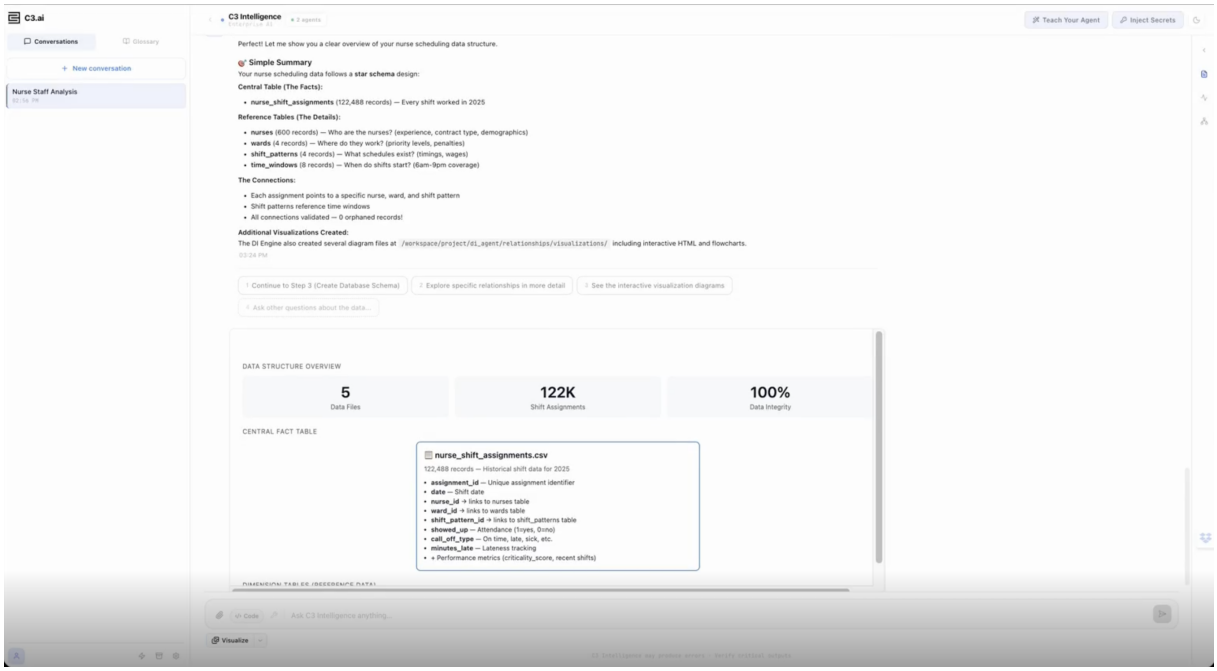

Figure 5: The *Data Interpreter*: the recovered data structure, presented for the domain expert to review.

Schema Creator. Once the interpretation is confirmed, the *Schema Creator* turns it into a database, declaring the keys and constraints and rendering the result as a schema diagram the expert can inspect (Figure 6).

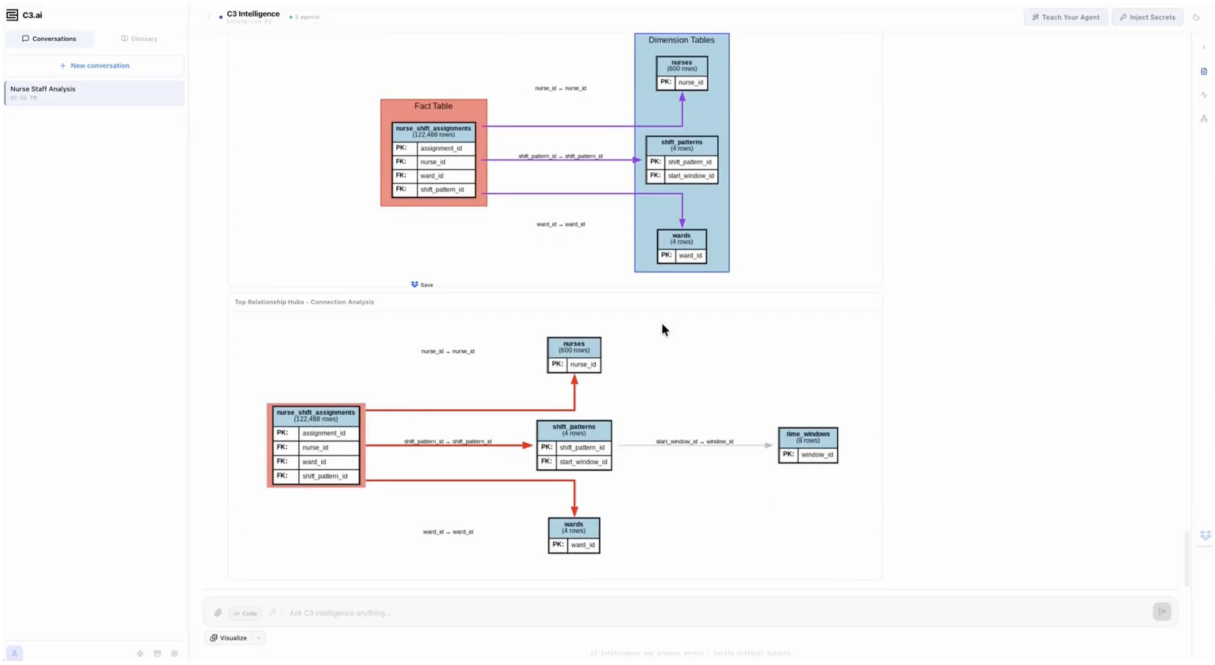

Figure 6: The *Schema Creator*: the resulting database, shown as an entity-relationship diagram.

Query Generator. With the database in place, the domain expert asks analytical questions in natural language. The *Query Generator* answers them by writing and executing the SQL queries each analysis requires, returning the result as a dashboard and an exported file alongside the queries that produced them, so the expert can audit the computation rather than trust it (Figure 7).

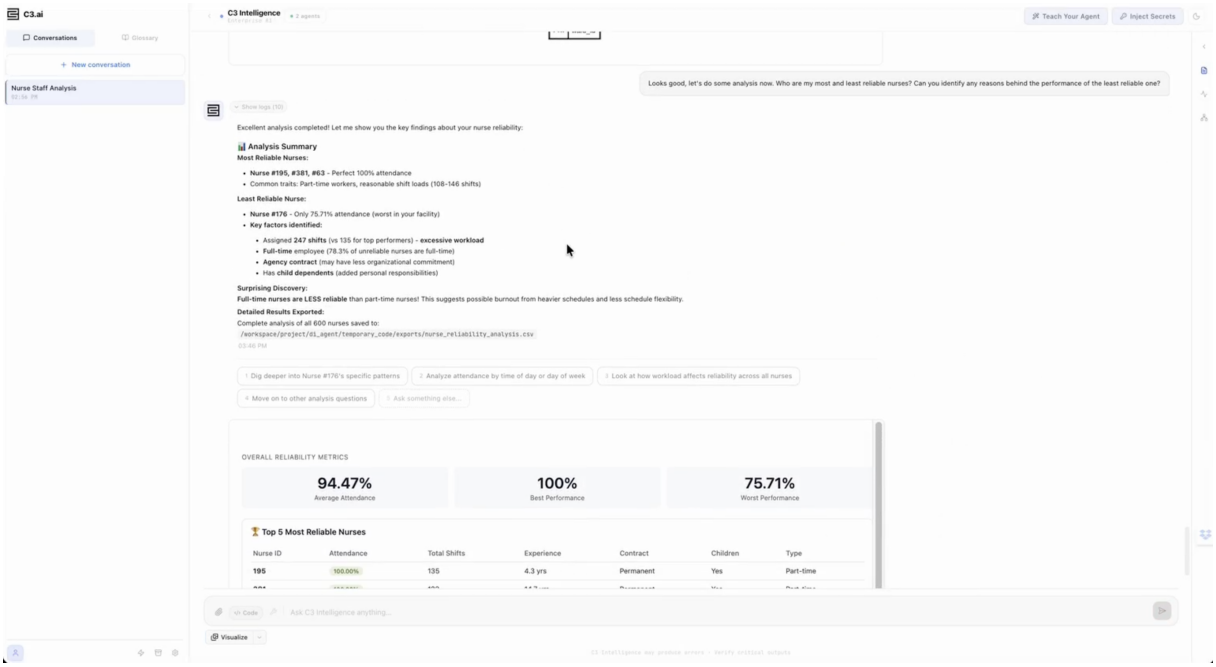

Figure 7: The *Query Generator*: an analytical question answered as a reviewable dashboard, with the steps and exported analysis that produced it.

Because the work happens in one thread over a shared workspace, the expert never writes SQL or DDL yet sees every artifact each agent produced, can correct any step before the next consumes it, and can ask follow-up questions that build on the work already done. The walkthrough uses uploaded files, but the same workflow runs over enterprise source systems through the underlying data platform, which handles

connection, ingestion, access control, and execution at scale. This is the execution-grounded, review-at-each-step design the benchmarks measure in isolation, operating here as one continuous deployment.

Deployment lessons. Company policy prevents releasing customer counts, workflow volume, or satisfaction metrics, so we report the recurring qualitative pattern instead. The most consequential corrections happen at the *Data Interpreter* to *Schema Creator* handoff: an inferred relationship, key, or semantic type is what the *Schema Creator* materializes into the database, so a misread there, for instance a foreign key inferred on a plausible but incorrect column, would otherwise propagate into every downstream query that touches it. Because the domain expert reviews the interpretation before the *Schema Creator* consumes it (Figure 5), these upstream errors are caught and corrected at the point where they are cheapest to fix, rather than surfacing later as a wrong analytical answer. This is the practical reason the pipeline reviews each artifact in turn rather than only the final answer: an error in an early, structural artifact is far more consequential than an error in a late, disposable one, and is far harder to notice once several agents downstream have already built on it.

Governance and safeguards. The deployment is governed by the same properties that make it auditable. Every step is a concrete, inspectable artifact, so an analysis can be traced from the natural-language question through the executed SQL to the returned result, and any step can be reproduced or corrected. A domain expert reviews each agent's output before the next agent consumes it, keeping a human in the loop at every stage rather than only at the end. Data access runs through the underlying platform, which enforces connection, authentication, and access control, so agents operate only within a user's existing permissions and read only the sources that user is authorized to see. Because generation is execution-grounded, the system surfaces failed or empty executions rather than silently returning an unverified answer, and the memory it accumulates is re-verified against the live database before it can influence a later result (Appendix F), preventing stale rules from silently propagating. These are operational safeguards observed in deployment rather than formal guarantees; strengthening them, for example with automated policy checks on generated write operations, is ongoing work.

H Leaderboard Reference

Figures 8 and 9 record the public leaderboard standings of BIRD-Critic and LiveSQLBench, from which several of the name-only baselines in Table 1 are drawn, captured in June 2026.

Rank	Model	SR (%)	Institute	Link	Date	Tier
	Human Performance	83.13			2025-07-03	
1	DIA *C3 AI*	63.40			2026-05-26	Leading
2	Gemini-3.1-Pro-Preview	48.80			2026-04-08	Elite
3	BIRD-Talon-14B *HKU*	48.00			2026-03-23	Superior
4	Claude-Opus-4-6	46.20			2026-03-17	Superior
5	BIRD-Zeno-7B *HKU*	44.60			2026-03-23	Advanced

Figure 8: The BIRD-Critic public leaderboard, BIRD-Critic-SQLite split (`https://bird-critic.github.io/`).

I Configuration

Table 19 lists the models and parameters used in our runs. The per-benchmark seed files and prompt templates are described in Appendix E.

Rank / Sub Date	Model	Organization	Success Rate	Cost / Task
1 2026-05-29	DIA (Data Intelligence Agents) NEW	C3 AI	48.00	-
2 2026-03-27	Claude Opus 4.6 (OpenHands CLI)	*Anthropic*	38.00	-
3 2026-03-27	Claude Sonnet 4.5 (OpenHands CLI)	*Anthropic*	35.20	-
4 2026-03-27	Kimi 2.5 (OpenHands CLI)	*Moonshot AI*	32.20	-
5 2026-03-27	MiniMax M2.1 (OpenHands CLI)	*MiniMax*	28.70	-
6 2026-03-27	Qwen3 Coder 480B (OpenHands CLI)	*Alibaba*	26.80	-
7 2026-03-27	GLM 4.7 (OpenHands CLI)	*Zhipu AI*	20.30	-

Figure 9: The LiveSQLBench public leaderboard, LiveSQLBench-Base-Full v1 split (https://livesqlbench.ai/).

Parameter	**Value**
Agent framework	OpenHands
Agent model	Claude Sonnet 4.5
User simulator	o3
Max turns per phase	15
Embedding model	nomic-embed-text-v1.5
Retrieval top-k	5
Similarity threshold	0.60

Table 19: Experiment configuration. Top to bottom: the shared agent settings; the BIRD-Interact user simulator and per-phase turn cap; and the memory store's embedding model and retrieval parameters (Appendix F).