

LYNX: Analytical Program Synthesis via Grounding and Closed-Loop Refinement

Ruichen Xia^{1,2*}, Xiao Han^{1*}, Dingyu Yang^{1†}, Luwei Yang³,
Sai Wu¹, Dongxiang Zhang¹, Binhui Yao^{1,3}, Gang Chen¹

¹ The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China

² Institute of Computing Innovation, Zhejiang University, Hangzhou, China

³ Zhejiang Geely Holding Group Co., Ltd., Hangzhou, China

rcxia@zju.edu.cn, 3230100653@zju.edu.cn, yangdingyu@zju.edu.cn, Luwei.Yang1@geely.com

wusai@zju.edu.cn, zhangdongxiang@zju.edu.cn, 12321292@zju.edu.cn, cg@zju.edu.cn

Abstract—LLM-powered data analytics aims to translate natural language instructions into executable analytical programs over underlying data. However, existing approaches—whether treating LLMs as intent-to-code translators or as standalone analytical engines—often fail to reliably align domain semantics with complex schemas and heterogeneous metadata, resulting in brittle execution and semantically inconsistent analytical outputs in real-world workloads. In this paper, we present LYNX, a multi-agent framework that formulates LLM-driven analytics as a schema- and knowledge-constrained analysis problem rather than a single-pass generation task. LYNX organizes analytical synthesis into a closed-loop workflow of planning, generation, verification, and refinement through three role-specialized agents that collaboratively enforce schema grounding, semantic consistency, and execution correctness. To address domain ambiguity and schema misalignment, LYNX integrates Semantic-Structural Distillation and Stage-Aware Allocation (SSD-SAA). Furthermore, a delta-guided iterative revision continuously diagnoses and repairs semantic and execution-level failures, substantially improving synthesis robustness. We evaluate LYNX on two NL-to-SQL benchmarks, BIRD-SQL Mini-Dev and more challenging Spider 2.0-Lite. Experimental results demonstrate strong performance and efficiency, achieving 10.66% to 59.54% execution accuracy across lightweight and commercial models, while reducing token consumption by 70.53% compared with the prior most efficient method. Also, our framework displays high generalization over different backbones, from open-source, lightweight models to the latest commercial models. Additional case studies on quantitative finance analytics and complex multi-relational SQL workloads further demonstrate the framework’s robustness, scalability, and practical applicability in real-world analytical scenarios. Our code is available at <https://github.com/summerhare12/LYNX>.

Index Terms—NL-to-SQL, Domain Semantics, Multi-agent System, Structured Chain-of-Thought, Iterative Self-correction

I. INTRODUCTION

LLM-powered data analytics aims to translate natural language analytical queries into executable analytical programs by jointly reasoning over query semantics, database schemas, metadata, and execution constraints [1]–[3]. This capability enables domain experts to express analytical intent through domain knowledge rather than low-level programming, making it particularly attractive for enterprise and scientific analytics involving complex workflows and heterogeneous datasets. However, reliably synthesizing executable analytical programs remains challenging in real-world environments with heterogeneous schemas, complex metadata, and domain-specific semantics. Even minor misalignment between natural language intent and structured data semantics can lead to execution failures or executable-but-incorrect analytical programs [4]–[6].

Existing approaches to LLM-powered data analytics mainly follow two directions. The first treats LLMs as *intent-to-code translators*, where models generate executable programs (e.g., SQL or Python) that are executed by external analytical engines. Representative systems include Text-to-SQL frameworks such as DIN-SQL [7], CHESS [8], and agent-based systems such as OpenAgents [9]. While these methods effectively leverage the code generation capability of LLMs, they typically rely on single-pass synthesis or static schema linking, making them vulnerable to weak schema grounding, execution failures, and semantically incorrect analytical programs when domain constraints are incompletely captured [10], [11]. The second direction employs LLMs as reasoning-centric analytical models that directly operate over tabular or structured data without explicit execution engines. Representative examples include Galois [12], TableGPT [13], and TableLLaMA [14]. Although these approaches reduce dependence on external systems, they often struggle with large-scale or multi-relational analytical workloads and are more susceptible to semantic,

* These authors contributed equally to this work.

† Corresponding author.

factual, and arithmetic reasoning errors compared with traditional database execution engines [15].

Despite their architectural differences, both paradigms ultimately struggle with two fundamental issues: accurately grounding domain semantics to database schemas and ensuring semantic correctness throughout analytical program execution. To address these limitations, we identify the following fundamental challenges:

C1: Domain and Schema Grounding. General-purpose LLMs often lack sufficient understanding of domain-specific terminology and the underlying database structures, including table schemas, attribute types, and inter-table relationships [16]–[19]. Consequently, models frequently produce incorrect schema linking, hallucinated attributes, inconsistent joins, or invalid type usage [20]–[24], leading to semantically incorrect or non-executable analytical programs.

C2: Execution Reliability and Semantic Consistency. Even when generated programs are syntactically valid, LLMs often struggle to preserve semantic consistency across multi-step analytical reasoning, including proper function usage, condition composition, and dependency preservation [10], [11], [25]–[27]. Such limitations produce executable-but-incorrect programs that satisfy syntactic and execution constraints while violating the intended analytical semantics. This highlights the inherent brittleness of single-pass analytical program synthesis and the need for explicit semantic verification.

To address these challenges, we propose LYNX, a framework that jointly integrates schema grounding, analytical reasoning, and semantic verification within a unified process. The framework consists of three coordinated components—**Generator**, **Critic**, and **Reviser**—that collaboratively perform grounded generation, execution-guided refinement, and intent-level validation.

To address **C1 (Domain and Schema Grounding)**, LYNX introduces **Semantic-Structural Distillation and Stage-Aware Allocation** that integrates conceptual semantics, structured metadata abstractions, and syntax specifications into the generation process. Then the decoupled knowledge is elaborately distilled and processed. LYNX dynamically allocates them to the corresponding reasoning stages, ensuring that grounding information is incorporated rather than merely appended to the prompt. This design reduces semantic ambiguity and enables more reliable alignment.

For the challenge of Execution Reliability and Semantic Consistency (**C2**), LYNX introduces an **execution-guided revision loop and verification mechanism**. Critic Agent performs delta-guided memory management across iterations, explicitly identifying error evolution and repair effectiveness while maintaining a compact memory. Reviser Agent then leverages this feedback

to perform targeted program repair and localized correction. Together, they form a tightly coupled feedback loop that progressively resolves execution errors. Beyond execution correctness, a final semantic verification stage explicitly validates consistency between the synthesized program and the original analytical intent, mitigating false positives that cannot be detected through execution feedback alone.

We evaluate LYNX on Spider 2.0-Lite and BIRD-SQL Mini-Dev. Experimental results demonstrate state-of-the-art performance across diverse backbone models while maintaining high token efficiency. On Spider 2.0-Lite, LYNX achieves up to **51.13%** execution accuracy, substantially outperforming prior methods under the same backbone settings. On BIRD-SQL Mini-Dev, it reaches **49.40%** execution accuracy. Moreover, LYNX requires only **11.2K–13.8K** tokens per task with three major backbones, substantially reducing inference cost while maintaining state-of-the-art accuracy. The framework generalizes effectively from lightweight open-source models (e.g., Qwen3-14B) to state-of-the-art commercial LLMs, and extensive ablation studies confirm the contribution of each major component.

In summary, our contributions are as follows:

- We propose LYNX, a framework for domain-constrained analytical program synthesis that unifies schema grounding, execution-guided refinement, and semantic verification to improve the reliability of LLM-powered analytics.
- We develop Semantic-Structural Distillation and Stage-Aware Allocation for knowledge grounding, a mechanism that refines domain semantics and schema knowledge and allocates them to the corresponding synthesis stages.
- We introduce an execution-guided refinement and semantic verification mechanism that performs delta-based failure diagnosis, targeted program repair, and intent-level validation to reduce executable-but-incorrect programs.
- Extensive experiments on Spider 2.0-Lite and BIRD-SQL Mini-Dev demonstrate state-of-the-art execution accuracy, superior token efficiency, and strong generalization across both open-source and commercial LLM backbones.

II. RELATED WORKS

Recent LLM-powered analytics systems mainly differ in whether LLMs serve as intent translators that orchestrate external analytical engines or directly act as analytical engines themselves [28], [29].

A. Code as Engine: Translators and Tool Users

This paradigm represents the most mature and widely adopted approach, where LLMs function as intent trans-

lators that convert natural language requests into executable artifacts, typically SQL queries or Python programs, which are then executed by reliable external engines such as DBMSs or interpreters. Its effectiveness is grounded in the strong code generation capabilities of modern LLMs [30], [31], and has been further extended by agent-based systems that enable tool use for multi-step analysis, including general platforms such as **OpenAgents** [9] and task-specific solutions in Text-to-SQL, e.g., divide-and-conquer parsing in **DIN-SQL** [7], multi-agent collaboration in **CHESS** [8], and prompt-level optimization in **DailSQL** [32].

Despite their effectiveness, these approaches remain vulnerable to execution failures and executable-but-incorrect programs caused by incomplete grounding or flawed reasoning [10], [11]. Moreover, most systems employ loosely coupled retrieval, generation, and verification stages, limiting the use of execution feedback during synthesis. Recent methods such as **ReFoRCE** [33] and **AutoLink** [34] improve reliability through candidate exploration, but at the cost of substantial token and inference overhead.

B. LLM as Engine: Direct Data Reasoner

This line of work explores a more nascent and experimental paradigm, where LLMs are elevated to act as direct analytical engines, reasoning over data without relying on external execution environments. Representative approaches follow two main directions: modular reasoning frameworks that decompose analytical tasks into predefined operators or reusable components [35], exemplified by systems such as **Galois** [12] and code-centric models that retrieve and adapt program fragments (e.g., **Codex** [36], **Code Llama** [37]); and table-specialized LLMs that aim to natively process and reason over tabular data, including **TableGPT** [13], [38], **TableLlama** [14], and **TableLLM** [15].

Despite their promise, these approaches lack the execution guarantees provided by traditional analytical engines, making them difficult to scale to large or multi-relational datasets and more susceptible to semantic, factual, and arithmetic errors during complex analytical reasoning [13], [14].

III. METHODOLOGY

The proposed framework, LYNX (Figure 1), formulates LLM-powered analytics as a *domain-constrained analytical program synthesis* problem. Rather than relying on single-pass code generation, LYNX introduces a unified framework that jointly addresses two fundamental challenges: *domain and schema grounding* and *execution reliability with semantic consistency*. The framework consists of three tightly coordinated components: Generator, Critic, and Reviser.

To address the grounding challenge, LYNX introduces a **Semantic-Structural Distillation and Stage-Aware Allocation (SSD-SAA)** mechanism. A key observation is that retrieving relevant knowledge does not necessarily imply effective utilization during program synthesis. SSD-SAA separates grounding information into semantic and structural knowledge, refines them into compact synthesis-oriented representations, and dynamically allocates them to the corresponding reasoning stages. By explicitly coordinating knowledge usage with intent understanding, schema grounding, and code generation, this design establishes stronger alignment among analytical intent, database schemas, and executable program structures.

To ensure execution reliability and semantic consistency, LYNX further adopts the self-revision and verification loop. The Critic performs **delta-guided memory management** across iterations and maintains a compact history representation of historical debugging signals, allowing the framework to identify error evolution and repair effectiveness. Guided by this feedback, the Reviser performs targeted program repair for syntax, schema, and logical errors. A final logical verification stage then validates consistency between the synthesized program and the original analytical intent through intent-level reasoning, mitigating executable-but-incorrect failures that cannot be detected through execution feedback alone.

These components transform analytical program synthesis from a single-pass language modeling task into a closed-loop process of grounding, diagnosis, repair, and verification, which substantially improves execution correctness, semantic fidelity, and robustness in complex domain-specific analytical workloads.

A. Generator Agent

The Generator is responsible for transforming analytical intent into an initial executable program through explicit domain and schema grounding. To this end, we introduce **Semantic-Structural Distillation and Stage-Aware Allocation**, which decomposes grounding into three stages: **Decoupled Knowledge Acquisition**, **Knowledge Refinement**, and **Stage-Aware Allocation**. SSD-SAA converts heterogeneous knowledge sources into synthesis-oriented representations and dispatches them to the reasoning stages where they are most useful.

1) *Decoupled Knowledge Acquisition*: LYNX adopts a **decoupled grounding strategy** that separates multi-source semantic knowledge into *Conceptual Semantics* and *Structural Schema* (Figure 2) and processes them independently based on their own unique characteristics.

First, for *Conceptual Semantics*, LYNX constructs a semantic grounding context through a hybrid strategy that combines dense knowledge in official documents with keyword-based matching, capturing both

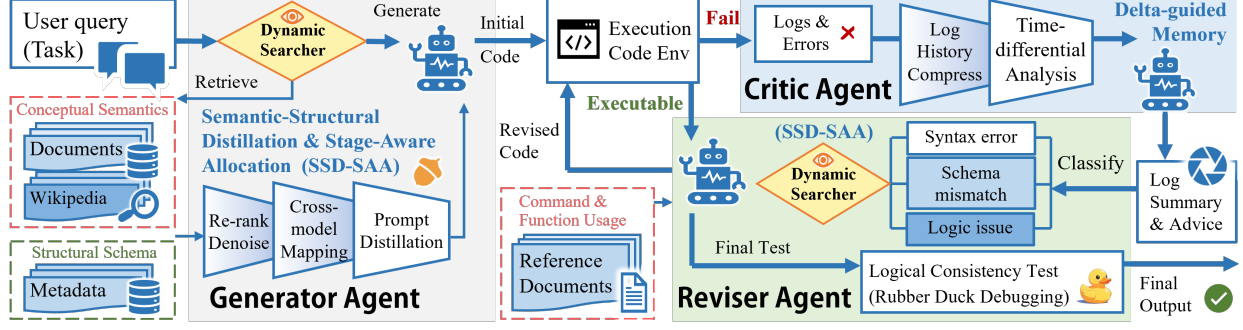


Fig. 1: The Framework of the Multi-agent System

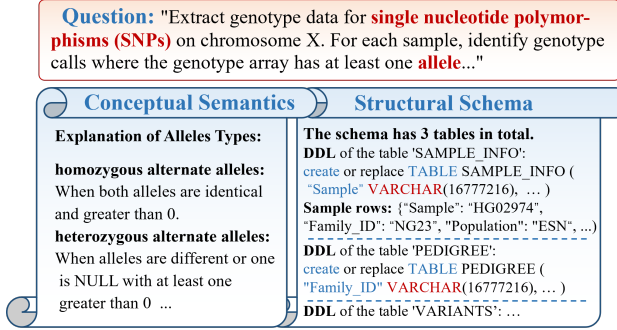


Fig. 2: Decoupled Domain Knowledge Example

domain-specific identifiers and analytical concepts. To improve robustness under domain drift and previously unseen terminology, we further introduce a **Just-in-Time semantic expansion** mechanism that selectively supplements insufficient retrieval results using external knowledge sources (e.g., Wikipedia). Once the LLM makes the judgment that its parametric knowledge and document information are insufficient, it will activate the mechanism to acquire supplementary semantics, which are subsequently filtered to remove noisy content before being incorporated into the synthesis context.

Then, for *Structural Schema*, LYNX performs schema grounding through dedicated **Metadata Abstraction**. Unlike conventional approaches that directly serialize raw metadata into prompts, the module extracts and normalizes heterogeneous metadata sources—including table schemas, column descriptions, data types, DDL statements, and JSON specifications—into a unified canonical representation. To reduce context redundancy, homogeneous structures (e.g., partitioned tables sharing identical schemas) are compactly summarized while preserving execution-critical constraints and inter-table relationships. The abstraction provides an execution-oriented view of the underlying organization and serves as the structural grounding source for program synthesis.

2) *Knowledge Refinement*: After acquisition, the two knowledge sources remain crude and cannot be directly consumed for staged synthesis. To bridge the gap between semantic concepts and executable program structures, SSD-SAA applies a progressive knowledge refinement process that transforms raw results into a compact, stage-dispatchable representation. Concretely, this process performs: (1) **Relevance-aware re-ranking**: Queries, documents, and schema descriptions are encoded by BGE-M3 and ranked by cosine similarity to retain the top- k evidence items. (2) **Semantic-structural alignment**: Retrieved concepts are linked to corresponding schema elements and operator specifications. (3) **Knowledge distillation**: The aligned knowledge is compressed into a compact synthesis-oriented representation that preserves essential constraints while minimizing context overhead.

We further integrate **fallback mechanism** to enhance robustness under real-world deployment conditions, as illustrated in Figure 3. Unpredictable or noisy web-retrieved content may trigger internal safety filters and interrupt normal generation. When it happens, the system automatically discards risky external text while preserving essential schema and metadata context, allowing the Generator agent to continue functioning. This design considers the ranking and priority of heterogeneous semantics, which ensures graceful degradation rather than complete failure.

3) *Stage-Aware Knowledge Allocation*: Rather than injecting all refined knowledge at once, SSD-SAA orchestrates synthesis as a stage-aware allocation process that decomposes program generation into three tightly scoped sub-tasks: semantic intent analysis, schema grounding, and code formulation. During semantic intent analysis, only the distilled conceptual knowledge K_{sem}^* is supplied to disambiguate analytical concepts and user objectives. During schema grounding, the distilled structural schemas K_{struct}^* is incorporated to map the resolved intent to concrete database elements. Finally, during code formulation, the operator-level constraints

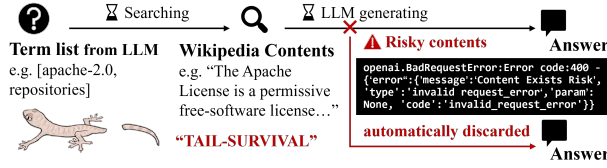


Fig. 3: Risks Contents Fallback Mechanism

Algorithm 1 SSD-SAA: Semantic-Structural Distillation and Stage-Aware Allocation (Generator Agent)

Require: Query Q , Documents Doc , Wikipedia $\mathcal{W}$, Schema $\mathcal{S}$, Syntax and Function Usage Syn

Ensure: Initial Code Generation $Code$

```

1: // 1. Decoupled Knowledge Acquisition
2:  $K_{sem} \leftarrow \text{HYBRIDRETRIEVE}(Q, Doc, W)$   $\triangleright$  Semantic concepts with JIT expansion
3:  $K_{struct} \leftarrow \text{SCHEMAABSTRACT}(\mathcal{S})$   $\triangleright$  Normalized structural representation
4: // 2. Knowledge Distillation & Alignment
5:  $K_{rel} \leftarrow \text{RERANK}(Q, K_{sem} \cup K_{struct})$   $\triangleright$  Relevance-aware filtering
6:  $K_{align} \leftarrow \text{ALIGN}(K_{rel}^{sem}, K_{rel}^{struct})$   $\triangleright$  Explicit concept-to-schema mapping
7:  $K^* \leftarrow \text{DISTILL}(K_{align})$   $\triangleright$  Compress into high-signal compact form
8:  $(K_{sem}^*, K_{struct}^*) \leftarrow \text{PARTITION}(K^*)$   $\triangleright$  Split knowledge by reasoning stage
9: // 3. Stage-Aware Allocation
10:  $I \leftarrow \text{INTENTANALYSIS}(Q, K_{sem}^*)$ 
11:  $G \leftarrow \text{SCHEMAGROUNDING}(I, K_{struct}^*)$ 
12:  $Code \leftarrow \text{FORMULATION}(G, K_{op}^*)$ 
13: return  $Code$ 

```

and syntax specifications from official references K_{syn}^* are applied to generate the executable analytical program. This explicit stage-conditioned allocation ensures that each category of knowledge is consumed precisely at the stage where it is most relevant, avoiding the inefficient utilization and weak grounding that plague undifferentiated knowledge injection. The resulting synthesis pipeline yields analytical programs with strong semantic fidelity and high execution reliability.

B. Critic Agent

Synthesizing executable and semantically faithful code remains a significant challenge for LLMs, particularly under complex analytical constraints. To address this issue, we design the Critic agent around a central principle: effective debugging requires not merely accumulating error logs, but maintaining **delta-guided memory management** that distills historical debugging experience and

explicitly analyzes error transitions to inform subsequent revisions (Algorithm 2).

1) **Memory Adaptive Reflection:** A naive iterative strategy that repeatedly appends all historical errors and code versions quickly leads to excessive context growth, degrading efficiency and even exceeding the model’s context window. To overcome this, the Critic implements this failure memory as an **adaptive reflection mechanism** that compactly summarizes past debugging data while preserving its diagnostic value. Specifically, the agent maintains a rolling error history updated via a reflect-and-append mechanism: instead of retaining raw logs, it semantically compresses prior iterations $t-1$ and appends only the latest code–error pair:

$$H_t = \text{Reflect}(H_{t-1}) \oplus (\text{Code}_t + \text{Error}_t)$$

This design introduces a principled **temporal decay**, analogous to momentum indicators in financial analysis [39], using a time-weighted scheme to prioritize recent performance while gracefully mitigating stale information. Concretely, the summarized **long-term memory** captures recurring error patterns and their proven corrections, while the **short-term focus** remains anchored on the latest code and its corresponding execution error. This dual-granularity structure balances long-term pattern retention with short-term precision, eliminating redundancy while maintaining comprehensive awareness of the debugging trajectory.

2) **Delta-guided Failure Diagnosis:** Beyond memory compression, the Critic agent performs explicit **delta-guided failure diagnosis** by systematically comparing error transitions across successive iterations. By analyzing code and error deltas between iterations $t-1$ and t , the agent distinguishes two complementary diagnostic signals:

- **Constructive progression:** E_t differs in nature from E_{t-1} . The previous correction (changes from $Code_{t-1}$ to $Code_t$) successfully resolved the prior issue. This positive delta is recorded in debugging history as a validated solution pattern that offers reusable lessons for future debugging.
- **Debugging stagnation:** E_t remains equivalent or closely related to E_{t-1} . This negative delta signals that the current solution path is ineffective, and the agent is explicitly instructed to abandon it in favor of a divergent strategic alternative.

The core mechanism is the delta examination, which explicitly compares changes in code and error messages between consecutive iterations. This produces a meta-level diagnostic that prevents repetitive failures and actively steers the revision cycle away from proven dead ends toward more promising solution spaces. By grounding revision guidance in error patterns rather

Algorithm 2 Delta-Guided Memory Management
(Critic Agent)

Require: Current $Code_t$, Current Error E_t , Previous History H_{t-1} , Previous $Code_{t-1}$, Previous Error E_{t-1}

Ensure: Updated History H_t , Suggestion S_t

```

1: // 1. Adaptive Reflection Mechanism
   (Temporal Decay)
2:  $H_s \leftarrow \text{Reflect}(H_{t-1})$   $\triangleright$  Distill long-term failure
   memory
3:  $H_t \leftarrow H_s \oplus (Code_t, E_t)$   $\triangleright$  Integrate current
   failure state
4: // 2. Delta-Guided Failure
   Diagnosis
5:  $\Delta E \leftarrow \text{Analyze}(E_{t-1}, E_t)$ 
6: if Error changes then  $\triangleright$  Positive delta: Fix works
7:    $S_t \leftarrow \text{Constructive Progression}$   $\triangleright$  Record as
   useful solution pattern in history
8: else  $\triangleright$  Negative delta: Fix failed
9:    $S_t \leftarrow \text{Debugging Stagnation}$   $\triangleright$  Explore
   divergent path
10: end if
11: return  $H_t, S_t$ 

```

than isolated error instances, the Critic provides more actionable suggestions for the subsequent Reviser agent.

Overall, the Critic agent maintains a distilled, delta-aware reasoning trace that audits the debugging process, prevents logical drift, and stabilizes the refinement trajectory toward correctness. Operating under explicit execution supervision, it leverages both the current code state and blocker feedback to provide precise, transition-informed guidance to the Reviser agent.

C. Reviser Agent

The Reviser is responsible for transforming partially correct programs into executable and semantically faithful programs. Unlike the Generator, which focuses on initial synthesis, the Reviser operates under execution supervision and leverages structured feedback from the Critic to perform targeted program repair and validation.

1) *Execution-Guided Program Repair*: The first responsibility of the Reviser is to correct execution failures. Given the delta-guided diagnosis generated by the Critic, the Reviser categorizes failures into three classes:

- **Syntax-level failures**, caused by invalid statements, incorrect function usage, or malformed expressions;
- **Schema-level failures**, caused by incorrect table references, attribute mismatches, or invalid schema linking;
- **Semantic-level failures**, caused by missing constraints, incorrect filtering conditions, or flawed analytical logic.

User Question Which customer has the highest average monthly change in payment amounts? Provide the customer's full name.	
The Code (Candidate SQL)	VS The Rubber Duck (Plain Text Explanation)
<pre> WITH monthly_totals AS (SELECT STRFTIME('%Y-%m', payment_date) AS month, FROM payment GROUP BY customer_id, payment_month), monthly_changes AS (SELECT total_amount - LAG(total_amount) OVER (PARTITION BY customer_id ORDER BY payment_month) AS amount_change FROM monthly_totals), avg_changes AS (SELECT ... FROM monthly_changes WHERE amount_change IS NOT NULL GROUP BY customer_id)...</pre>	The SQL first computes the total payment amount per customer per month using STRFTIME. It then calculates the month-to-month change in total amount for each customer by subtracting the previous month's total (obtained via LAG) from the current month's total. For each customer, it averages these monthly changes (ignoring nulls from the first month ...). Finally, it joins the ... and orders by the average monthly change in descending order, returning only the top customer.
Implicit Logic (points to <code>STRFTIME</code>) Complex CTE (points to <code>monthly_changes</code>) Null Handling Issues (points to <code>WHERE amount_change IS NOT NULL</code>)	Step-by-Step Translation (points to the explanation text) Verify User Constraints (points to the final ordering) NULL issues (points to the <code>WHERE</code> clause)
Analysis: The plain English translation confirms the SQL correctly interprets average monthly change by calculating consecutive differences and then averaging per customer. It correctly handles edge cases (single month data) and returns the requested fields for the top customer. Decision: Approved (The logic is flawless. Output unchanged.)	

Fig. 4: Rubber Duck Debugging Strategy

For each failure category, the Reviser selectively consults the corresponding grounding information provided by the same SSD-SAA module, including domain knowledge, schema metadata, and operator specifications. Rather than regenerating the entire program, the Reviser performs localized repair by identifying and modifying only the failure-inducing code segments. The revised program is then re-executed, forming an execution-guided refinement loop.

2) *Intent-Level Semantic Verification*: Execution success does not necessarily imply analytical correctness. In practice, many generated programs satisfy execution constraints while violating the original analytical intent. To mitigate such executable-but-incorrect failures, the Reviser performs an additional semantic verification stage after execution succeeds.

A direct self-evaluation of generated code often suffers from confirmation bias, where the model tends to validate its own outputs [40], [41]. To address this issue, we adopt a **Rubber Duck Debugging** strategy as an intent-level verification mechanism illustrated in Figure 4. Specifically, the Reviser first translates the generated program into a structured natural-language explanation, explicitly exposing its analytical logic, including filtering conditions, aggregation operations, and output requirements. The explanation is then systematically compared against the original user request to verify semantic consistency.

When inconsistencies are detected, the Reviser triggers another round of targeted correction. Otherwise, the program is accepted as the final output. By combining execution-guided repair with intent-level semantic verification, the Reviser effectively reduces both execution failures and executable-but-incorrect programs.

TABLE I: Comparison of performance results on Spider 2.0-Lite

Model	Method	Easy	Medium	Hard	Total EX	Avg. Tokens
GPT-4o	DIN-SQL	5.79	0.43	0.00	1.46	—
	CHESS	9.92	3.00	1.24	3.84	—
	DAIL-SQL	13.20	5.58	1.24	5.68	—
	LYNX (ours)	44.36	37.25	22.14	34.05	13.8K
DeepSeek-R1	RSL-SQL	—	—	—	30.53	50.0K
	LinkAlign	—	—	—	33.09	—
	AutoLink	—	—	—	34.92	38.0K
	LYNX (ours)	68.00	53.08	35.92	51.13	11.2K
DeepSeek-V3	Spider-Agent	20.31	6.10	4.05	8.78	—
	LinkAlign	—	—	—	24.86	—
	RSL-SQL	—	—	—	26.14	—
	LYNX (ours)	45.75 $\pm$ 0.72	38.35 $\pm$ 0.89	23.97 $\pm$ 1.76	35.45$\pm$1.02	13.4K

IV. EXPERIMENT

A. Experimental Setup

1) *Testbed*: To ensure fair and interpretable comparisons, we adopt **GPT-4o**, **DeepSeek-V3**, and **DeepSeek-R1** as the primary backbones, as they are commonly used in recent NL-to-SQL and LLM-agent benchmarks. For all settings, the decoding temperature is fixed to 0 to ensure reproducibility. To study backbone sensitivity and generalization, we further evaluate LYNX on a diverse set of models spanning both open-source and proprietary LLMs, including **Qwen3-14B**, **Claude-Sonnet-4.6**, and **Gemini-3.1-Pro**.

2) *Workloads*: We evaluate LYNX on two widely used text-to-SQL benchmarks: **BIRD-SQL Mini-Dev** [42] and **Spider 2.0-Lite** [43]. BIRD-SQL Mini-Dev contains 500 instances derived from realistic databases and domain-specific analytical scenarios, while Spider 2.0-Lite contains 547¹ instances spanning BigQuery, Snowflake, and SQLite environments. Spider 2.0-Lite further emphasizes realistic and complex analytical workflows, with all SQL queries exceeding 50 whitespace-tokenized tokens and categorized into three difficulty levels based on token count: easy (<80 tokens), medium (80~159 tokens), and hard ($\geq$ 160 tokens). Together, these benchmarks evaluate both semantic grounding and execution reliability in real-world analytical tasks.

3) *Evaluation Metrics*: Following the official evaluation protocols of BIRD-SQL Mini-Dev and Spider 2.0-Lite [43], [44], we use **Execution Accuracy (EX)** as the primary metric, which measures whether the generated analytical program produces the correct execution result.

For BIRD-SQL Mini-Dev, we additionally report **Valid Efficiency Score (VES)** and **Soft-F1**. VES jointly evaluates execution correctness and efficiency, while Soft-F1 measures the semantic similarity between

the predicted and ground-truth result sets at the cell level [44]. Together, these metrics provide a comprehensive evaluation of analytical correctness, execution efficiency, and result-level semantic fidelity.

4) *Baselines*: We compare LYNX against representative methods spanning prompt-based, agent-based, and retrieval-enhanced text-to-SQL systems. **Prompt-based methods** include **DIN-SQL** [45], **DAIL-SQL** [32], and **CHESS** [8], which improve SQL generation through task decomposition, demonstration selection, and structured prompting. **Agent-based systems** include **Spider-Agent** [43], which performs iterative query construction through tool interactions. **Recent state-of-the-art systems** include **RSL-SQL** [46], **ReFoRCE** [33], **LinkAlign** [47], and **AutoLink** [34], which incorporate schema grounding, retrieval enhancement, iterative refinement, and candidate exploration mechanisms to improve robustness on complex analytical workloads.

B. Main Results

1) *Overall Performance*: Table I reports the results on Spider 2.0-Lite. Following standard practice in recent NL-to-SQL literature, we report baseline numbers from their original papers or official releases. To ensure fairness, LYNX is evaluated using the same backbone models adopted by the corresponding baselines, allowing the comparison to focus on framework design rather than model capability. Overall, **LYNX consistently achieves the highest execution accuracy across all evaluated backbones while maintaining substantially lower token consumption**. With GPT-4o, LYNX achieves an EX of **34.05%**, significantly outperforming DAIL-SQL (5.68%). Under DeepSeek-V3, LYNX reaches an average of **35.45%** ($\pm$ 1.02), exceeding LinkAlign (24.86%) and RSL-SQL (26.14%), while the low variance across repeated runs demonstrates stable performance. When instantiated with DeepSeek-R1, LYNX achieves the highest EX of **51.13%**, outper-

¹Task sf009, sf013 and sf029 report database unavailable.

TABLE II: LYNX performance on different backbones

Backbone Model	EX %	Avg. Times	Avg. Tokens
Qwen3-14B	10.66	6.50	48.4K
GPT-4o	34.05	5.16	13.8K
DeepSeek-V3	35.45	4.25	13.4K
DeepSeek-R1	51.13	4.80	11.2K
Claude-Sonnet-4.6	57.58	4.29	7.1K
Gemini-3.1-Pro	59.54	3.70	6.4K

forming AutoLink (34.92%) and RSL-SQL (30.53%). The gains are particularly evident on complex queries. We attribute this improvement to the complementary effects of SSD-SAA and the delta-guided refinement loop. SSD-SAA strengthens domain and schema grounding during synthesis, while the Critic-Reviser feedback loop progressively eliminates execution and semantic errors, reducing the brittleness of single-pass generation and improving analytical correctness under complex reasoning workloads.

2) *Token Efficiency*: We measure token consumption as the cumulative prompt and completion tokens used per task until the framework produces an executable program that passes semantic verification. Since token usage grows with the number of generation and revision rounds, it serves as a practical indicator of inference cost. As shown in Table I, LYNX requires only **11.2K–13.8K** tokens per task with three major backbones. In comparison, AutoLink and RSL-SQL consume **38.0K** and **50.0K** tokens, respectively. This substantial reduction in token consumption stems from our execution-guided repair strategy, which incrementally refines a single candidate program rather than relying on expensive candidate exploration, enabling LYNX to achieve both higher accuracy and lower inference cost.

3) *Generalization Across Models*: Table II evaluates LYNX across a diverse set of backbone LLMs, ranging from lightweight open-source models to recent frontier models. Overall, LYNX generalizes well across a broad spectrum of model capabilities and consistently benefits from advances in stronger foundation models. With frontier models such as **DeepSeek-R1**, **Claude-Sonnet-4.6**, and **Gemini-3.1-Pro**, LYNX consistently achieves more than **50%** EX while requiring fewer refinement iterations, resulting in lower token consumption. This trend indicates that stronger backbones can more effectively leverage the grounding and execution-guided refinement mechanisms provided by LYNX.

At the same time, the framework remains effective on lightweight models. As shown in Table II, **Qwen3-14B** still achieves **10.66%** EX, outperforming several existing systems built upon substantially stronger commercial backbones. This result suggests that the performance gains of LYNX are not solely derived from backbone ca-

TABLE III: Comparison of Executability Performances

Metric	DeepSeek-V3	Qwen3-14B
Executability	88.42%	49.36%
Avg. Attempts	4.25	6.50
Few-Shot (≤ 3 attempts)	68%	43%
Executable Rate		

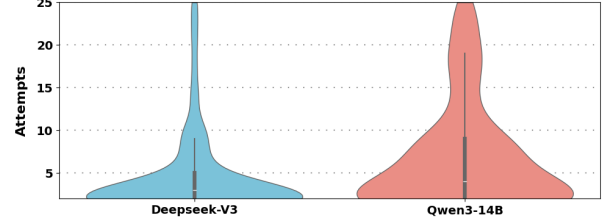


Fig. 5: Attempt Distribution of Models

pability, but also from its underlying synthesis architecture. Nevertheless, model capacity remains an important factor in determining the performance ceiling. Compared with Qwen3-14B, DeepSeek-V3 achieves substantially higher final executability (**88.42%** vs. **49.36%**) and requires fewer refinement iterations (**4.25** vs. **6.50**), indicating that stronger models can utilize execution feedback more effectively. Figure 5 further shows that a large proportion of Qwen3-14B’s successful queries are obtained only after **6–10** refinement rounds. This observation suggests that execution-guided refinement is particularly valuable for lightweight models, where a considerable fraction of initially incorrect programs can still be repaired through iterative feedback.

4) *Effect on BIRD-SQL Mini-Dev*: Table IV reports the results on BIRD-SQL Mini-Dev, and LYNX consistently achieves the best performance across all evaluation metrics. With DeepSeek-V3, it attains **49.40%** EX, **45.44%** VES, and **56.31%** Soft-F1, outperforming previous state-of-the-art methods. The simultaneous improvements in EX, VES, and Soft-F1 indicate that LYNX not only generates more correct analytical programs but also produces higher-quality execution results with better efficiency. The gains are particularly notable when using lightweight models. When instantiated with **Qwen3-14B**, LYNX achieves **43.40%** EX, surpassing TA [37], [48] built upon the substantially larger Llama3-70B backbone. This result suggests that the effectiveness of LYNX is not solely dependent on model scale, but also benefits from its grounding and execution-guided refinement mechanisms. Compared with Spider 2.0-Lite, BIRD places greater emphasis on domain-specific semantics and realistic analytical scenarios. The results suggest that accurate grounding via SSD-SAA and iterative refinement via the Critic-Reviser loop jointly drive the performance gains. A detailed breakdown across

TABLE IV: Performance Comparison on BIRD Mini-Dev

Methods (Models)	EX (%)	VES	Soft-F1
TA + codellama-34B [37]	27.57	-	-
GPT-3.5-turbo [49]	38.00	37.33	41.84
llama3-70B-instruct	40.80	39.02	44.38
Claude-2 [50]	42.70	-	-
TA + llama3-70B-instruct [48]	42.80	41.37	46.66
TA + DeepSeek [48]	43.74	-	-
Struct-SQL [51]	45.00	-	-
LYNX(ours) + Qwen3-14B	43.40	40.48	49.92
LYNX(ours) + Deepseek-V3	49.40	45.44	56.31

TABLE V: Detailed Results across Difficulty Levels

Model	Metric	Simple	Moderate	Hard	Total
Qwen3-14B	EX	57.43	39.20	33.33	43.40
	VES	54.11	36.10	31.44	40.48
	Soft-F1	62.95	45.22	42.55	49.92
DeepSeek-V3	EX	63.51	47.20	34.31	49.40
	VES	58.69	43.02	32.14	45.44
	Soft-F1	68.22	54.91	42.50	56.31

difficulty levels is provided in Table V.

5) *Dialect Adaptability*: To understand the impact of dialect-specific syntax and schema characteristics, we further evaluate LYNX across multiple SQL dialects. Figure 6 presents the corresponding execution accuracy of **SQLite**, **BigQuery**, **Snowflake**. SQLite achieves the highest accuracy due to its relatively standardized syntax and simpler schema organization. In contrast, performance degrades on BigQuery and Snowflake, which introduce dialect-specific operators, proprietary functions, and more complex schema abstractions. The gap is particularly pronounced on Snowflake, where analytical programs often involve hierarchical object references (e.g., `DATABASE.SCHEMA.TABLE`), specialized semi-structured data operators, and dialect-specific function semantics. These characteristics increase the difficulty of both schema grounding and syntax generation, making execution correctness more challenging to achieve. Despite these challenges, LYNX maintains consistent performance advantages across all dialects. The remaining performance gap indicates that dialect-specific reasoning remains a challenging problem for current LLMs and represents a promising direction for future improvement.

C. Ablation Analysis

1) *Agent Ablation*: Table VI reports the ablation results using DeepSeek-V3. Since the Generator, Critic, and Reviser form a tightly coupled synthesis pipeline, we perform ablation at the **functional level** rather than removing entire agents. Overall, all components contribute substantially to the final performance, confirming the effectiveness of the proposed closed-loop design.

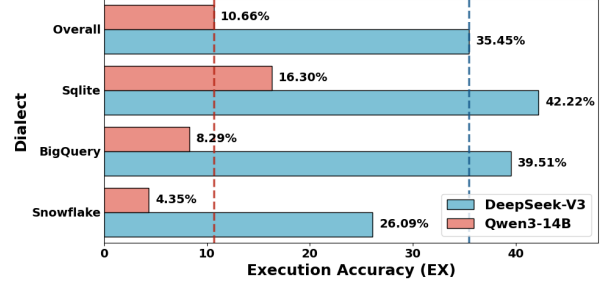


Fig. 6: Scores on Different SQL Dialects

TABLE VI: Ablation Results on Spider 2.0-lite

agent	function	EX % (Δ)
-	Full model (Ours)	35.45
Generator	w/o SSD-SAA	≈ 0 ($\downarrow \approx 100\%$)
Critic	w/o Delta-guided memory	27.36 ($\downarrow 22.8\%$)
Reviser	w/o Iterative revision	22.43 ($\downarrow 36.7\%$)
	w/o Logical verification	30.23 ($\downarrow 14.7\%$)

Effect of Generator. Removing SSD-SAA causes execution accuracy to collapse to nearly zero. Without explicit semantic and schema grounding, the model cannot reliably resolve domain terminology, schema elements, or structural constraints, making analytical program synthesis largely infeasible. This result highlights that accurate grounding is a prerequisite for complex NL-to-analytics tasks.

Effect of Critic. Removing the Critic reduces EX from **35.45%** to **27.36%**. Without differential error analysis and reflection memory, the framework loses visibility into error evolution across iterations, making it difficult to distinguish effective repairs from repeated failures. This demonstrates the importance of execution-aware diagnosis for guiding iterative refinement.

Effect of Reviser. Disabling execution-guided repair decreases EX from **35.45%** to **22.43%**, indicating that a substantial portion of generated programs require targeted correction before becoming executable. Furthermore, removing semantic verification reduces EX to **30.23%**, showing that execution success alone does not guarantee analytical correctness. The additional improvement confirms the effectiveness of intent-level verification in mitigating executable-but-incorrect programs.

Overall, the results validate the complementary roles of the three components: Generator provides reliable grounding, the Critic supplies execution-aware diagnostic signals, and the Reviser progressively improves both execution correctness and semantic fidelity through iterative refinement.

TABLE VII: Token Consumption and Runtime Analysis

Agent	Tokens		Runtime
	Prompt	Completion	
Generator	3520 $\pm$ 627	496 $\pm$ 102	14.94 $\pm$ 4.08s
Critic	3462 $\pm$ 1128	934 $\pm$ 228	6.82 $\pm$ 1.40s
Reviser	2438 $\pm$ 413	473 $\pm$ 113	10.16 $\pm$ 3.92s

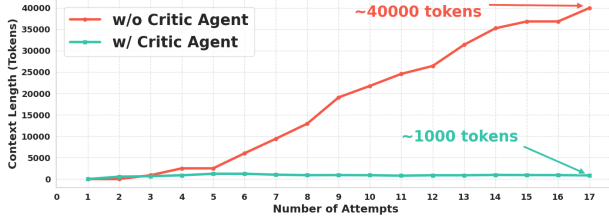


Fig. 7: Comparison of History Length after Attempts

2) *Overhead Analysis*: Table VII reports the token consumption and runtime breakdown of the three components. The overhead remains moderate and aligns with the functional responsibilities of each component. The **Generator** incurs the highest runtime (14.94 s/query) and prompt cost, as it performs knowledge acquisition, refinement, and stage-aware grounding before synthesis. The **Critic** exhibits the largest completion length because it maintains compressed error traces and produces diagnostic feedback for subsequent repair. In contrast, the **Reviser** introduces the lowest prompt overhead, as it operates on structured diagnostic signals rather than raw execution contexts. A key concern in iterative program synthesis is the growth of historical debugging context. To evaluate the effectiveness of the Critic’s memory management mechanism, we track the length of `error_history` across revision iterations (Figure 7). Without compression, the accumulated debugging context grows approximately linearly and can exceed **40K** tokens, rapidly becoming the dominant source of inference cost. In contrast, the Critic maintains a compact representation through differential summarization, effectively bounding context growth while preserving diagnostic information.

3) *Stage-aware Reasoning Efficacy*: Stage-aware reasoning is a core design principle of LYNX and is employed throughout the Generator, Critic, and Reviser. To evaluate its contribution, we remove all task decomposition and stage-specific reasoning instructions. As shown in Figure 8, stage-aware reasoning consistently improves performance across different backbones and query difficulty levels, indicating that it provides a general benefit rather than targeting a specific query category. The improvement is particularly evident on DeepSeek-V3 (+6.59% EX), compared with Qwen3-14B (+1.84% EX), suggesting that stronger models

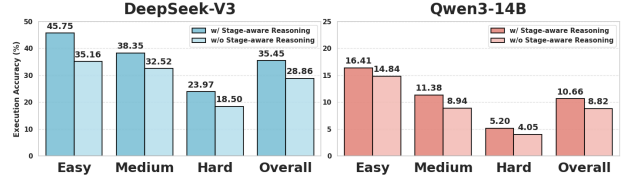


Fig. 8: Details for Stage-aware Reasoning Efficacy

TABLE VIII: Revision Loop Effects across Models

Setting \ Model	Qwen3-14B	DeepSeek-V3
	EX (%)	EX (%)
w/o Revision	4.96	22.43
w/ Revision	10.66	35.45
Δ	5.70 ($\uparrow 115\%$)	13.02 ($\uparrow 58\%$)

are better able to exploit stage-aware reasoning and grounding signals during analytical program synthesis.

4) *Iterative Revision Loop*: To quantify the impact of execution-guided refinement, Table VIII compares the performance with and without the iterative revision loop. The results demonstrate substantial gains across models of different scales. For Qwen3-14B, execution accuracy more than doubles from **4.96%** to **10.66%**. Under DeepSeek-V3, EX increases from **22.43%** to **35.45%**. These improvements indicate that a significant fraction of initially generated programs can be repaired through iterative execution feedback. Notably, the relative gains are larger for Qwen3-14B, suggesting that execution-guided refinement is particularly valuable for lightweight models, where it compensates for weaker schema understanding and domain knowledge through progressive self-correction.

5) *BIRD-SQL Complementary Results*: Tables IX reports the impact of semantic grounding on BIRD-SQL Mini-Dev. Compared with Spider 2.0-Lite, BIRD adopts relatively homogeneous SQLite schemas, making many schema structures and operators easier to infer from the model’s parametric knowledge. Consequently, semantic grounding yields smaller gains in execution accuracy. Nevertheless, LYNX consistently improves result quality, increasing **Soft-F1** by **2.32** (specifically **+2.00** for hard subset). This suggests that semantic grounding remains beneficial even when execution accuracy saturates, as it improves semantic alignment between generated programs and analytical intent. The results further indicate that the effectiveness of grounding mechanisms becomes more pronounced as schema complexity and domain ambiguity increase.

V. CASE STUDY

While aggregate metrics quantify overall performance, they provide limited visibility into how individual components contribute to analytical program synthesis. We

TABLE IX: Semantic Impact to Subsets on BIRD

Metric	Method	Difficulty			
		Simple	Moderate	Hard	Total
EX	w/o Semantics	62.16	44.00	36.27	47.80
	w/ Semantics	63.51	47.20	34.31	49.40
	Δ	+1.35	+3.20	-1.96	+1.60
VES	w/o Semantics	56.74	40.17	34.23	43.86
	w/ Semantics	58.69	43.02	32.14	45.44
	Δ	+1.95	+2.85	-2.09	+1.58
Soft-F1	w/o Semantics	65.83	52.48	40.50	53.99
	w/ Semantics	68.22	54.91	42.50	56.31
	Δ	+2.39	+2.43	+2.00	+2.32

present two representative case studies to examine the behavior of LYNX under realistic analytical workloads. The first case study focuses on a quantitative finance task and illustrates how the Critic–Reviser loop identifies execution failures, tracks error evolution, and progressively repairs analytical programs. The second case study is drawn from Spider 2.0-Lite and demonstrates how SSD-SAA grounds domain semantics and schema information to support complex multi-table reasoning.

A. Quantitative Finance Workloads

Quantitative finance presents a challenging setting for analytical program synthesis due to its specialized terminology, complex operator semantics, and multi-step numerical reasoning. Many financial concepts (e.g., *ADV*, *VWAP*, and industry-neutralized factors) are under-represented in the pretraining corpora of general-purpose LLMs, while factor computation often involves rolling-window statistics, cross-sectional ranking, and nested arithmetic operations.

To evaluate whether LYNX generalizes beyond text-to-SQL benchmarks, we apply it to the computation of Alpha#36 from the widely used **WorldQuant 101 Alphas** [52]. The workload consists of CSV data for **100** stocks, each containing more than **3,900** daily observations and 33 financial attributes. In addition to raw data, formal definitions of financial indicators and domain-specific operators are provided as external knowledge (Figure 9).

As illustrated in Figure 10, SSD-SAA first grounds financial terminology and operator semantics using the supplied domain knowledge. The Generator then synthesizes a multi-stage Python workflow covering data loading, preprocessing, intermediate feature computation, and factor aggregation, while the Critic–Reviser loop iteratively resolves execution and logical errors during synthesis. The resulting pipeline successfully computes Alpha#36 over the complete dataset. Figure 11 further shows the resulting factor distribution across stocks and time periods, indicating that the generated program produces meaningful and non-trivial financial

Input data: (“{ }” stands for a placeholder)	Functions and operators:
open, close, high, low, volume = standard definitions for daily data vwap = daily volume-weighted average price cap = market cap adv{d} = average daily dollar volume for the past d days 	abs(x), log(x), sign(x) = standard definitions; same for +, -, *, /, >, <, == rank(x) = cross-sectional rank delay(x, d) = value of x d days ago correlation(x, y, d) = time-serial correlation of x and y for the past d days delta(x, d) = today’s value of x minus the value of x d days ago

Fig. 9: Definitions for External Knowledge

signals rather than degenerate outputs. This demonstrates that LYNX can effectively combine domain grounding and iterative refinement to support complex numerical analytics beyond SQL generation.

B. Spider 2.0-Lite Case

To complement the aggregate benchmark results, we examine a representative Spider 2.0-Lite instance (*bq282*) that highlights the challenges of real-world analytical program synthesis. The task involves heterogeneous data containing mixed types and requires strict adherence to BigQuery-specific conventions. Even when powered by a frontier model (**GPT-5.2**), the generated query fails to execute due to incorrect handling of BigQuery **Fully Qualified Names (FQN)**, resulting in invalid table references. Furthermore, the query omits the required **safety casting** operations, making it vulnerable to malformed or missing values commonly encountered in real-world datasets.

Figure 13 illustrates how LYNX resolves this failure through execution-guided refinement. Instead of treating execution errors as isolated events, the Critic performs differential analysis over successive error states and identifies the underlying causes of failure. Guided by these diagnostic signals, the Reviser incrementally repairs the query by correcting dialect-specific table references and introducing appropriate type-safe transformations. Through multiple refinement iterations, the generated program evolves from a non-executable query into a valid solution that satisfies both execution and semantic requirements.

A side-by-side comparison between the failed GPT-5.2 output and the final query generated by LYNX is shown in Table X. This example demonstrates that strong language modeling capability alone is insufficient for reliable analytical program synthesis in realistic database environments. Instead, robust performance requires execution-aware diagnosis and iterative repair, which are explicitly supported by the Critic–Reviser feedback loop. The case therefore provides qualitative evidence for the effectiveness of the proposed execution-guided refinement mechanism beyond what can be observed from aggregate benchmark metrics alone.

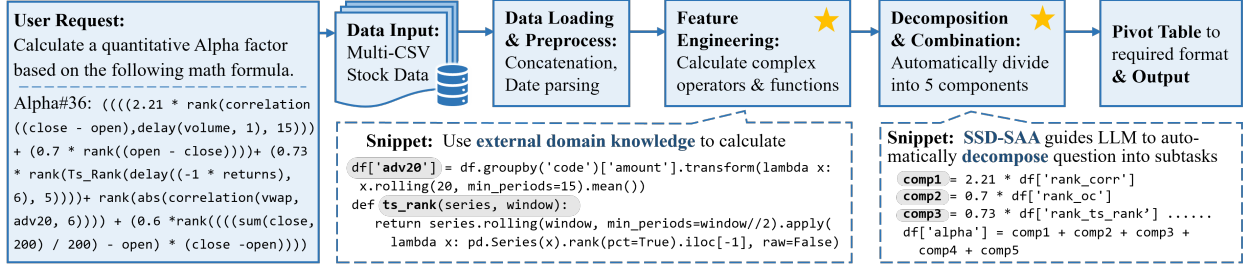


Fig. 10: The Pipeline of the Generated Python Script

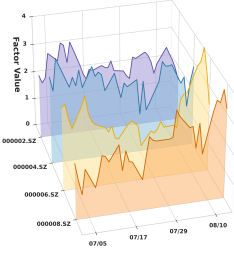


Fig. 11: Factor Results

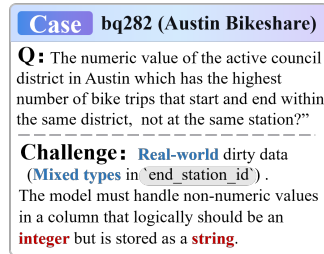


Fig. 12: Case Overview

TABLE X: Comparative Analysis of Case "bq282"

Stage	general-purpose LLM (GPT-5.2)	Our Framework (w/ revision)
Schema	Hallucinate table name	use correct FQN
Challenge	Standard casting fails on dirty data	Detected error and applied SAFE_CAST
Logic	Implicit type mismatch in comparisons	Explicitly compare INT64 vs STRING
Outcome	Execution Error "404 Not Found" ❌	Successfully executed Correct Answer ✅

Attempt 1 Initial Generation (The Trap)

- The model generates the standard query, assuming 'end_station_id' is clean strings. **Error: "Bad int64 value"** ❌

Attempt 2 Self-Correction (The Partial Fix)

- Reviser Agent:** identifies 'Event' as a non-numeric outlier. It introduces SAFE_CAST to handle the dirty data. ❌
- New Error:** "No matching signature for operator != (INT64, STRING)."

Attempt 3 Final Resolution (Success)

- Critic Agent:** correctly diagnoses the type collision caused by the previous fix.
- Reviser Agent:** explicitly casts both sides of the operands to 'INT64', ensuring type safety during comparison.
- Outcome:** Query executes successfully. ✅

Fig. 13: Case Multi-step Process

VI. CONCLUSION

In this paper, we present LYNX, a multi-agent framework that formulates reliable LLM-powered analytics as a domain-constrained analytical program synthesis process. LYNX addresses two central challenges in analytical program generation: grounding analytical intent in domain semantics and database structure, and reliably refining generated programs toward both execution correctness and semantic consistency. To address these challenges, LYNX integrates Semantic-Structural Distillation and Stage-Aware Allocation (SSD-SAA) for domain and schema grounding with a Critic-Reviser

feedback loop for iterative diagnosis, targeted repair, and intent-level semantic verification. SSD-SAA refines semantic and structural knowledge and allocates it to the corresponding synthesis stages, while the feedback loop uses execution outcomes and diagnostic state transitions to progressively repair candidate programs. Together, these mechanisms turn analytical program generation from one-shot translation into a closed-loop process in which grounding, execution, diagnosis, repair, and verification are explicitly coordinated. Our quantitative evaluation on Spider 2.0-Lite and BIRD-SQL Mini-Dev shows that LYNX achieves strong execution accuracy across multiple backbone models, SQL dialects, and query complexities. Under directly comparable settings, LYNX also improves token efficiency over existing iterative approaches. Additional analyses demonstrate the contributions of grounding, iterative refinement, and semantic verification, while the quantitative-finance case study illustrates the potential transfer of the framework beyond SQL-based analytics.

ACKNOWLEDGMENT

This work was supported by the "Pioneer" R&D Program of Zhejiang (2026C01019), and it was also supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62572422. GPT-5.6 Sol was used to assist with language polishing, grammar enhancement, and stylistic refinement of the manuscript.

REFERENCES

- [1] J. Jiang, H. Xie, S. Shen, Y. Shen, Z. Zhang, M. Lei, Y. Zheng, Y. Li, C. Li, D. Huang *et al.*, “Siriusbi: A comprehensive llm-powered solution for data analytics in business intelligence,” *Proceedings of the VLDB Endowment*, vol. 18, no. 12, pp. 4860–4873, 2025.
- [2] S. Minaee, T. Mikolov, N. Nikzad, M. Chenaghlu, R. Socher, X. Amatriain, and J. Gao, “Large language models: A survey,” 2024.
- [3] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong, Y. Pan, S. Xu, Z. Wu, Z. Liu, X. Zhang, S. Zhang, X. Hu, T. Zhang, N. Qiang, T. Liu, and B. Ge, “Understanding llms: A comprehensive overview from training to inference,” 2024.
- [4] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, “Emergent abilities of large language models,” 2022.
- [5] Z. Guo, R. Jin, C. Liu, Y. Huang, D. Shi, Supryadi, L. Yu, Y. Liu, J. Li, B. Xiong, and D. Xiong, “Evaluating large language models: A comprehensive survey,” 2023.
- [6] G. L. Xuanhe Zhou, Zhaoyan Sun, “Db-gpt: Large language model meets database,” 2023.
- [7] M. Pourreza and D. Rafiei, “Din-sql: Decomposed in-context learning of text-to-sql with self-correction,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 36 339–36 348, 2023.
- [8] S. Talaei, M. Pourreza, Y.-C. Chang, A. Mirhoseini, and A. Saberi, “Chess: Contextual harnessing for efficient sql synthesis,” *arXiv preprint arXiv:2405.16755*, 2024.
- [9] T. Xie, F. Zhou, Z. Cheng, P. Shi, L. Weng, Y. Liu, T. J. Hua, J. Zhao, Q. Liu, C. Liu *et al.*, “Openagents: An open platform for language agents in the wild,” *arXiv preprint arXiv:2310.10634*, 2023.
- [10] Z. Wang, Z. Zhou, D. Song, Y. Huang, S. Chen, L. Ma, and T. Zhang, “Towards understanding the characteristics of code generation errors made by large language models,” in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2025, pp. 717–717.
- [11] —, “Where do large language models fail when generating code?” *arXiv e-prints*, pp. arXiv–2406, 2024.
- [12] M. Saeed, N. De Cao, and P. Papotti, “Querying large language models with sql,” *arXiv preprint arXiv:2304.00472*, 2023.
- [13] L. Zha, J. Zhou, L. Li, R. Wang, Q. Huang, S. Yang, J. Yuan, C. Su, X. Li, A. Su *et al.*, “Tablegpt: Towards unifying tables, nature language and commands into one gpt,” *arXiv preprint arXiv:2307.08674*, 2023.
- [14] T. Zhang, X. Yue, Y. Li, and H. Sun, “Tablellama: Towards open large generalist models for tables,” in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2024, pp. 6024–6044.
- [15] X. Zhang, S. Luo, B. Zhang, Z. Ma, J. Zhang, Y. Li, G. Li, Z. Yao, K. Xu, J. Zhou *et al.*, “Tablellm: Enabling tabular data manipulation by llms in real office usage scenarios,” in *Findings of the Association for Computational Linguistics: ACL 2025*, 2025, pp. 10 315–10 344.
- [16] R. Cao, F. Lei, H. Wu, J. Chen, Y. Fu, H. Gao, X. Xiong, H. Zhang, W. Hu, Y. Mao *et al.*, “Spider2-v: How far are multimodal agents from automating data science and engineering workflows?” *Advances in Neural Information Processing Systems*, vol. 37, pp. 107 703–107 744, 2024.
- [17] Y. Zhang, J. Deriu, G. Katsogiannis-Meimarakis, C. Kosten, G. Koutrika, and K. Stockinger, “Sciencebenchmark: A complex real-world benchmark for evaluating natural language to sql systems,” *arXiv preprint arXiv:2306.04743*, 2023.
- [18] X. Liu, S. Shen, B. Li, P. Ma, R. Jiang, Y. Zhang, J. Fan, G. Li, N. Tang, and Y. Luo, “A survey of nl2sql with large language models: Where are we, and where are we going?” *arXiv preprint arXiv:2408.05109*, 2024.
- [19] X. Wang, X. Ling, K. Li, G. Yin, L. Zhang, J. Wu, A. Wang, and W. Wang, “Llm and agent-driven data analysis: A systematic approach for enterprise applications and system-level deployment,” *arXiv preprint arXiv:2511.17676*, 2025.
- [20] B. Wang, R. Shin, X. Liu, O. Polozov, and M. Richardson, “Rat-sql: Relation-aware schema encoding and linking for text-to-sql parsers,” in *Proceedings of the 58th annual meeting of the association for computational linguistics*, 2020, pp. 7567–7578.
- [21] M. Kothiyari, D. Dhingra, S. Sarawagi, and S. Chakrabarti, “Crush4sql: Collective retrieval using schema hallucination for text2sql,” *arXiv preprint arXiv:2311.01173*, 2023.
- [22] H. Ma, J. Zhu, M. R.-T. Lyu, and I. King, “Bridging the semantic gap between image contents and tags,” *IEEE Transactions on Multimedia*, vol. 12, no. 5, pp. 462–473, 2010.
- [23] K. Vaidya, A. Sankararaman, J. Ding, C. Lei, X. Qin, B. Narayanaswamy, and T. Kraska, “Odin: A nl2sql recommender to handle schema ambiguity,” *arXiv preprint arXiv:2505.19302*, 2025.
- [24] M. Pourreza, D. Rafiei, Y. Feng, R. Li, Z. Fan, and W. Zhang, “Sql-encoder: Improving nl2sql in-context learning through a context-aware encoder,” *arXiv preprint arXiv:2403.16204*, 2024.
- [25] J. Shen, C. Wan, R. Qiao, J. Zou, H. Xu, Y. Shao, Y. Zhang, W. Miao, and G. Pu, “A study of in-context-learning-based text-to-sql errors,” *arXiv preprint arXiv:2501.09310*, 2025.
- [26] M. Li, J. Chen, M. Xu, and X. Wang, “Hallucination detection in structured query generation via llm self-debating,” in *Findings of the Association for Computational Linguistics: EMNLP 2025*, 2025, pp. 16 102–16 113.
- [27] X. Xiong, S. Han, Z. Zhou, and A. Cohan, “Metamath: Integrating natural language and code for enhanced mathematical reasoning in large language models,” *arXiv e-prints*, pp. arXiv–2409, 2024.
- [28] X. Wang, F. Cheng, Y. Wang, K. Xu, J. Long, H. Lu, and H. Qu, “Interactive data analysis with next-step natural language query recommendation,” *arXiv preprint arXiv:2201.04868*, 2022.
- [29] P. Maddigan and T. Susnjak, “Chat2vis: Generating data visualizations via natural language using chatgpt, codex and gpt-3 large language models,” *Ieee Access*, vol. 11, pp. 45 181–45 193, 2023.
- [30] M. Chen, “Evaluating large language models trained on code,” *arXiv preprint arXiv:2107.03374*, 2021.
- [31] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, W.-t. Yih, D. Fried, S. Wang, and T. Yu, “Ds-1000: A natural and reliable benchmark for data science code generation,” in *International Conference on Machine Learning*. PMLR, 2023, pp. 18 319–18 345.
- [32] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian, B. Ding, and J. Zhou, “Text-to-sql empowered by large language models: A benchmark evaluation,” *arXiv preprint arXiv:2308.15363*, 2023.
- [33] M. Deng, A. Ramachandran, C. Xu, L. Hu, Z. Yao, A. Datta, and H. Zhang, “Reforce: A text-to-sql agent with self-refinement, consensus enforcement, and column exploration,” *arXiv preprint arXiv:2502.00675*, 2025.
- [34] Z. Wang, Y. Zheng, Z. Cao, X. Zhang, Z. Wei, P. Fu, Z. Luo, W. Chen, and X. Bai, “Autolink: Autonomous schema exploration and expansion for scalable schema linking in text-to-sql at scale,” *arXiv preprint arXiv:2511.17190*, 2025.
- [35] C. C. Sartori and C. Blum, “Combinatorial optimization for all: Using llms to aid non-experts in improving optimization algorithms,” *arXiv preprint arXiv:2503.10968*, 2025.
- [36] C. I. Chang, W. C. Choi, and I. C. Choi, “A systematic literature review of the opportunities and advantages for aigc (openai chatgpt, copilot, codex) in programming course,” in *Proceedings of the 2024 7th International Conference on Big Data and Education*, 2024, pp. 29–35.
- [37] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, “Code llama: Open foundation models for code,” *arXiv preprint arXiv:2308.12950*, 2023.
- [38] P. Li, Y. He, D. Yashar, W. Cui, S. Ge, H. Zhang, D. Rifin-ski Fainman, D. Zhang, and S. Chaudhuri, “Table-gpt: Table fine-

- tuned gpt for diverse table tasks,” *Proceedings of the ACM on Management of Data*, vol. 2, no. 3, pp. 1–28, 2024.
- [39] Z. Liu, S. Lu, and S. Wang, “Asymmetry, tail risk and time series momentum,” *International Review of Financial Analysis*, vol. 78, p. 101938, 2021.
- [40] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching large language models to self-debug,” *arXiv preprint arXiv:2304.05128*, 2023.
- [41] T. Elvira, T. T. Procko, J. O. Couder, and O. Ochoa, “Digital rubber duck: Leveraging large language models for extreme programming,” in *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. IEEE, 2023, pp. 295–304.
- [42] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, “Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [43] F. Lei, J. Chen, Y. Ye, R. Cao, D. Shin, H. Su, Z. Suo, H. Gao, W. Hu, P. Yin *et al.*, “Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows,” *arXiv preprint arXiv:2411.07763*, 2024.
- [44] J. Li, B. Hui, G. Qu, J. Yang, B. Li, B. Li, B. Wang, B. Qin, R. Geng, N. Huo *et al.*, “Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls,” *Advances in Neural Information Processing Systems*, vol. 36, pp. 42 330–42 357, 2023.
- [45] M. Pourreza and D. Rafiei, “Din-sql: Decomposed in-context learning of text-to-sql with self-correction,” *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [46] Z. Cao, Y. Zheng, Z. Fan, X. Zhang, and W. Chen, “Rsl-sql: Robust schema linking in text-to-sql generation,” *arXiv preprint arXiv:2411.00073*, 2024.
- [47] Y. Wang, P. Liu, and X. Yang, “Linkalign: Scalable schema linking for real-world large-scale multi-database text-to-sql,” in *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, 2025, pp. 977–991.
- [48] G. Qu, J. Li, B. Li, B. Qin, N. Huo, C. Ma, and R. Cheng, “Before generation, align it! a novel and effective strategy for mitigating hallucinations in text-to-sql generation,” *arXiv preprint arXiv:2405.15307*, 2024.
- [49] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray *et al.*, “Training language models to follow instructions with human feedback,” *Advances in neural information processing systems*, vol. 35, pp. 27 730–27 744, 2022.
- [50] P. Anthropic, “Introducing claude,” *March*, vol. 14, p. 2023, 2023.
- [51] K. Thaker and Y. Bresler, “Knowledge distillation with structured chain-of-thought for text-to-sql,” *arXiv preprint arXiv:2512.17053*, 2025.
- [52] Z. Kakushadze, “101 formulaic alphas,” *Wilmott*, vol. 2016, no. 84, pp. 72–81, 2016.
- [53] Y. Gao, Y. Xiong, X. Gao, K. Jia, J. Pan, Y. Bi, Y. Dai, J. Sun, H. Wang, and H. Wang, “Retrieval-augmented generation for large language models: A survey,” *arXiv preprint arXiv:2312.10997*, vol. 2, no. 1, 2023.
- [54] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, “A survey on rag meeting llms: Towards retrieval-augmented large language models,” in *Proceedings of the 30th ACM SIGKDD conference on knowledge discovery and data mining*, 2024, pp. 6491–6501.
- [55] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, “Language models are few-shot learners,” *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.
- [56] O. Topsakal and T. C. Akinci, “Creating large language model applications utilizing langchain: A primer on developing llm apps fast,” in *International conference on applied engineering and natural sciences*, vol. 1, no. 1, 2023, pp. 1050–1056.
- [57] M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou, “The faiss library,” *IEEE Transactions on Big Data*, 2025.
- [58] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, “Chain-of-thought prompting elicits reasoning in large language models,” *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.
- [59] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, “Large language models are zero-shot reasoners,” *Advances in neural information processing systems*, vol. 35, pp. 22 199–22 213, 2022.
- [60] B. Qin, B. Hui, L. Wang, M. Yang, J. Li, B. Li, R. Geng, R. Cao, J. Sun, L. Si *et al.*, “A survey on text-to-sql parsing: Concepts, methods, and future directions,” *arXiv preprint arXiv:2208.13629*, 2022.
- [61] S. Pahune and M. Chandrasekharan, “Several categories of large language models (llms): A short survey,” *arXiv preprint arXiv:2307.10188*, 2023.
- [62] T. Liu, F. Wang, and M. Chen, “Rethinking tabular data understanding with large language models,” in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2024, pp. 450–482.
- [63] J. Gardner, J. C. Perdomo, and L. Schmidt, “Large scale transfer learning for tabular data via language modeling,” *Advances in Neural Information Processing Systems*, vol. 37, pp. 45 155–45 205, 2024.

VII. APPENDIX

A. Challenge Details

As described in the Introduction, LLM analytical program synthesis mainly faces two challenges: **Domain and Schema Grounding, Execution Reliability and Semantic Consistency**.

The figure 14 illustrates these two challenges. **General-purpose LLM** produces an incorrect SQL snippet. It attempts to join `visits`, `transactions`, and `products` tables using inconsistent columns (`visitor_id`, `visit_id`, `product_id`) and lacks proper handling of nested structures (e.g., `UNNEST` on `hits` and `products`). This directly exemplifies **C1**: the model fails to understand the actual BigQuery schema of the `google_analytics_sample` dataset, where purchase events are stored within repeated `hits` and `product` fields. As a result, the query is not only semantically wrong but also non-executable. **Our Framework** demonstrates the correct approach: using `ga_sessions_201707*`, unnesting the fields of `hits` and `product`, filtering by `eCommerceAction.action_type = '6'` (completed purchase), and applying a `LOWER(product.v2ProductName) LIKE '%youtube%'` condition. This respects the schema and maintains semantic integrity.

Thus, the figure highlights both challenges: (1) lack of schema awareness leads to syntactically flawed joins and missing unnest operations (C1); (2) even if a query were executable, preserving correct multi-step logic—such as distinguishing purchasers from all visitors, excluding

specific product name patterns, and aggregating quantities—requires careful semantic consistency (C2), which single-pass LLMs often violate.

B. SSD-SAA Details

Retrieval-augmented generation has emerged as a pivotal technique for mitigating common limitations of LLMs, such as hallucinations and outdated internal knowledge [53], [54]. By dynamically integrating external knowledge sources, this mechanism enhances the factual accuracy of generated content and enables continuous knowledge updates without retraining the underlying model.

Conceptual Semantics: Our design integrates **multiple knowledge sources**, employs dense search for better semantic matching, and incorporates pre-processing (e.g., chunking and indexing) as well as refinement steps such as similarity ranking and filtering.

The Spider 2.0 benchmark provides external knowledge **documents** for part of its questions [43]. This is presented in **markdown documents** that define relevant domain terms, thereby offering the tested agent explanations for the questions. However, most questions are not paired with such external knowledge. In the **Spider 2.0-lite** dataset, which contains 547 examples, only 107 are accompanied by documentation, resulting in a coverage rate of 19.56%.

To compensate for the frequent lack of external knowledge, we introduce a Wikipedia searcher to supplement content. This can aid the LLM in better understanding tasks and deeper reasoning [30], [55].

We first prompt the LLM to read the question and identify specialized terms it does not understand. The LLM is instructed to output these terms as a comma-separated list (e.g., *"vaccine site, American Community Survey"* from the question: *"Given the latest population estimates from the 2018 five-year American Community Survey, what is the number of vaccine sites per 1000 people for counties in California?"*). Subsequently, the **WikipediaLoader** tool from LangChain [56] is utilized to retrieve and download the entries for reference. **Meta-data Parsing:** While Spider 2.0-lite provides local Meta-data, the complete dataset is primarily hosted on cloud database systems such as BigQuery and Snowflake [16].

To handle the large volume of external documents (e.g., Wikipedia entries), we pre-process the text through a split-and-embed pipeline. Documents are first segmented into coherent chunks to minimize irrelevant content interference, then embedded and indexed using the **FAISS** (Facebook AI Similarity Search) vectorstore [57]. FAISS enables scalable and efficient approximate nearest neighbor search, which is essential for querying large-scale knowledge bases. During inference, the user's natural language question is encoded into a dense vector,

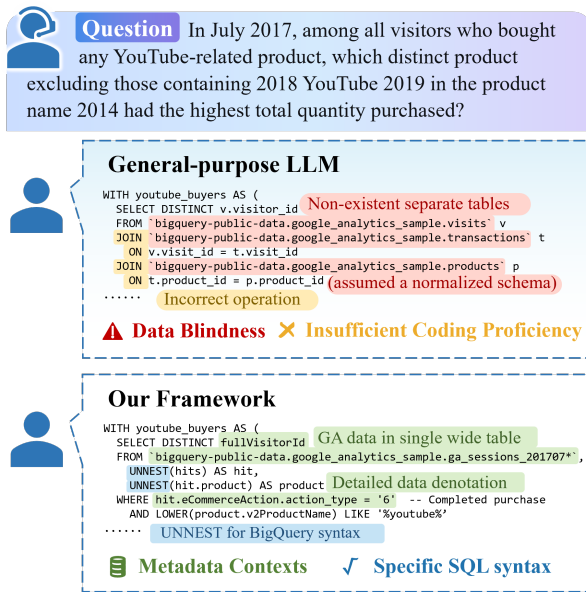


Fig. 14: An Example of Challenges

and a similarity search retrieves the top-k most relevant text passages.

Structural Schema: The Metadata for each database consists of two main components: Data Definition Language (DDL) statements in CSV files and a series of JSON summaries for every table. Each JSON summary details the table name, its Fully Qualified Name (FQN), column names, column data types, corresponding descriptions, and sample rows.

A notable challenge arises because many databases contain numerous homogeneous tables, which share an identical structure. For instance, the database `bigquery-public-data.google_analytics_sample` comprises 366 tables (e.g., `ga_sessions_20160923` through `ga_sessions_20170127`). To reduce informational redundancy and shorten the prompt length, we designed a specialized Metadata parsing mechanism. For homogeneous tables (e.g., those differentiated only by dates), the metadata is integrated, and only one representative table is described in detail. For heterogeneous tables, information for each table is presented and organized in a unified format.

A snippet of metadata parsing summary of the Snowflake database `_1000_GENOMES` is displayed as follows:

The schema has 3 tables in total.

SAMPLE_INFO DDL

```
1 create or replace TABLE SAMPLE_INFO (
2   "Sample" VARCHAR(16777216),
3   "Family_ID" VARCHAR(16777216),
4   "Population" VARCHAR(16777216),
5   "Population_Description" VARCHAR
6     (16777216),
7   "Gender" VARCHAR(16777216),
8   "Relationship" VARCHAR(16777216),
9   "Unexpected_Parent_Child" VARCHAR
10    (16777216),
11  "Non_Paternity" VARCHAR(16777216),
12  "Siblings" VARCHAR(16777216),
13  "Grandparents" VARCHAR(16777216),
14  "Avuncular" VARCHAR(16777216),
15  "Half_Siblings" VARCHAR(16777216),
16  "Unknown_Second_Order" VARCHAR(16777216),
17  "Third_Order" VARCHAR(16777216),
18  "In_Low_Coverage_Pilot" BOOLEAN,
19  "LC_Pilot_Platforms" VARCHAR(16777216),
20  "LC_Pilot_Centers" VARCHAR(16777216),
21  "In_High_Coverage_Pilot" BOOLEAN,
22  "HC_Pilot_Platforms" VARCHAR(16777216),
23  "HC_Pilot_Centers" VARCHAR(16777216),
24  "In_Exon_Targetted_Pilot" BOOLEAN,
25  "ET_Pilot_Platforms" VARCHAR(16777216),
26  "ET_Pilot_Centers" VARCHAR(16777216),
27  "Has_Sequence_in_Phase1" BOOLEAN,
28  "Phase1_LC_Platform" VARCHAR(16777216),
29  "Phase1_LC_Centers" VARCHAR(16777216),
30  "Phase1_E_Platform" VARCHAR(16777216),
31  "Phase1_E_Centers" VARCHAR(16777216),
32  "In_Phase1_Integrated_Variant_Set"
33    BOOLEAN,
```

```
31 "Has_Phase1_chrY_SNPs" BOOLEAN,
32 "Has_phase1_chrY_Deletions" BOOLEAN,
33 "Has_phase1_chrMT_SNPs" BOOLEAN,
34 "Main_project_LC_Centers" VARCHAR
35   (16777216),
36 "Main_project_LC_platform" VARCHAR
37   (16777216),
38 "Total_LC_Sequence" FLOAT,
39 "LC_Non_Duplicated_Aligned_Coverage"
40   FLOAT,
41 "Main_Project_E_Centers" VARCHAR
42   (16777216),
43 "Main_Project_E_Platform" VARCHAR
44   (16777216),
45 "Total_Exome_Sequence" FLOAT,
46 "X_Targets_Covered_to_20x_or_greater"
47   FLOAT,
48 "VerifyBam_E_Omni_Free" FLOAT,
49 "VerifyBam_E_Affy_Free" FLOAT,
50 "VerifyBam_E_Omni_Chip" FLOAT,
51 "VerifyBam_E_Affy_Chip" FLOAT,
52 "VerifyBam_LC_Omni_Free" FLOAT,
53 "VerifyBam_LC_Affy_Free" FLOAT,
54 "VerifyBam_LC_Omni_Chip" FLOAT,
55 "VerifyBam_LC_Affy_Chip" FLOAT,
56 "LC_Indel_Ratio" FLOAT,
57 "E_Indel_Ratio" FLOAT,
58 "LC_Passed_QC" BOOLEAN,
59 "E_Passed_QC" BOOLEAN,
60 "In_Final_Phase_Variant_Calling" BOOLEAN,
61 "Has_Omni_Genotypes" BOOLEAN,
62 "Has_Axiom_Genotypes" BOOLEAN,
63 "Has_Affy_6_0_Genotypes" BOOLEAN,
64 "Has_Exome_LOF_Genotypes" BOOLEAN,
65 "EBV_Coverage" FLOAT,
66 "DNA_Source_from_Coriell" VARCHAR
67   (16777216),
68 "Has_Sequence_from_Blood_in_Index"
69   BOOLEAN,
70 "Super_Population" VARCHAR(16777216),
71 "Super_Population_Description" VARCHAR
72   (16777216)
73 );
```

Sample Row:

```
1 {
2   "Sample": "HG02974",
3   "Family_ID": "NG23",
4   "Population": "ESN",
5   "Population_Description": "Esan_in_
6     Nigeria",
7   "Gender": "female",
8   "Relationship": "mother",
9   "Unexpected_Parent_Child": "",
10  "Non_Paternity": "",
11  "Siblings": "",
12  "Grandparents": "",
13  "Avuncular": "",
14  "Half_Siblings": "",
15  "Unknown_Second_Order": "",
16  "Third_Order": "",
17  "In_Low_Coverage_Pilot": null,
18  "LC_Pilot_Platforms": "",
19  "LC_Pilot_Centers": "",
20  "In_High_Coverage_Pilot": null,
21  "HC_Pilot_Platforms": "",
22  "HC_Pilot_Centers": "",
```

```

22 "In_Exon_Targetted_Pilot": null,
23 "ET_Pilot_Platforms": "",
24 "ET_Pilot_Centers": "",
25 "Has_Sequence_in_Phase1": null,
26 "Phase1_LC_Platform": "",
27 "Phase1_LC_Centers": "",
28 "Phase1_E_Platform": "",
29 "Phase1_E_Centers": "",
30 "In_Phase1_Integrated_Variant_Set": null,
31 "Has_Phase1_chrY_SNPs": null,
32 "Has_phase1_chrY_Deletions": null,
33 "Has_phase1_chrMT_SNPs": null,
34 "Main_project_LC_Centers": "BI",
35 "Main_project_LC_platform": "ILLUMINA",
36 "Total_LC_Sequence": 32595580924.0,
37 "LC_Non_Duplicated_Aligned_Coverage":
    10.17,
38 "Main_Project_E_Centers": "BI",
39 "Main_Project_E_Platform": "ILLUMINA",
40 "Total_Exome_Sequence": 7576615180.0,
41 "X_Targets_Covered_to_20x_or_greater":
    0.917355,
42 "VerifyBam_E_Omni_Free": 0.00487,
43 "VerifyBam_E_Affy_Free": 0.00541,
44 "VerifyBam_E_Omni_Chip": null,
45 "VerifyBam_E_Affy_Chip": 0.00543,
46 "VerifyBam_LC_Omni_Free": 0.00263,
47 "VerifyBam_LC_Affy_Free": 0.0038,
48 "VerifyBam_LC_Omni_Chip": null,
49 "VerifyBam_LC_Affy_Chip": 0.00391,
50 "LC_Indel_Ratio": 1.3,
51 "E_Indel_Ratio": 1.09,
52 "LC_Passed_QC": true,
53 "E_Passed_QC": true,
54 "In_Final_Phase_Variant_Calling": true,
55 "Has_Omni_Genotypes": null,
56 "Has_Axiom_Genotypes": null,
57 "Has_Affy_6_0_Genotypes": true,
58 "Has_Exome_LOF_Genotypes": null,
59 "EBV_Coverage": 92.29,
60 "DNA_Source_from_Coriell": "LCL",
61 "Has_Sequence_from_Blood_in_Index": null,
62 "Super_Population": "AFR",
63 "Super_Population_Description": "African"
64 }

```

PEDIGREE DDL

```

1 create or replace TABLE PEDIGREE (
2   "Family_ID" VARCHAR(16777216),
3   "Individual_ID" VARCHAR(16777216),
4   "Paternal_ID" VARCHAR(16777216),
5   "Maternal_ID" VARCHAR(16777216),
6   "Gender" NUMBER(38,0),
7   "Phenotype" NUMBER(38,0),
8   "Population" VARCHAR(16777216),
9   "Relationship" VARCHAR(16777216),
10  "Siblings" VARCHAR(16777216),
11  "Second_Order" VARCHAR(16777216),
12  "Third_Order" VARCHAR(16777216),
13  "Other_Comments" VARCHAR(16777216)
14 );

```

VARIANTS DDL

```

1 create or replace TABLE VARIANTS (
2   "reference_name" VARCHAR(16777216),

```

```

3   "start" NUMBER(38,0),
4   "end" NUMBER(38,0),
5   "reference_bases" VARCHAR(16777216),
6   "alternate_bases" VARIANT,
7   "quality" FLOAT,
8   "filter" VARIANT,
9   "names" VARIANT,
10  "call" VARIANT,
11  AA VARCHAR(16777216),
12  AC VARIANT,
13  AC1 NUMBER(38,0),
14  AF FLOAT,
15  AF1 FLOAT,
16  AFR_AF FLOAT,
17  AMR_AF FLOAT,
18  AN NUMBER(38,0),
19  ASN_AF FLOAT,
20  AVGPOST FLOAT,
21  CIEND VARIANT,
22  CIPOS VARIANT,
23  DP NUMBER(38,0),
24  DP4 VARIANT,
25  ERATE FLOAT,
26  EUR_AF FLOAT,
27  FQ FLOAT,
28  G3 VARIANT,
29  HOMLEN NUMBER(38,0),
30  HOMSEQ VARCHAR(16777216),
31  HWE FLOAT,
32  LDAF FLOAT,
33  MQ NUMBER(38,0),
34  PV4 VARIANT,
35  RSQ FLOAT,
36  SNPSOURCE VARIANT,
37  SOURCE VARIANT,
38  SVLEN NUMBER(38,0),
39  SVTYPE VARCHAR(16777216),
40  THETA FLOAT,
41  VT VARCHAR(16777216)
42 );

```

These passages, together with the parsed Metadata, are assembled into an augmented prompt, as **Input-Layer Integration** [54]. Retrieved Wikipedia passages and documentation are combined with the parsed metadata and incorporated into the agent's prompt. This process synthesizes a comprehensive background for the task, enabling the agent to ground its interpretation in relevant external knowledge and significantly reduce schema and terminological hallucinations, guiding it to generate more accurate and schema-aware SQL queries [55].

Stage-aware Allocation: To enhance stage-aware reasoning, the agent follows a carefully designed prompting strategy. By providing the modularized reasoning decomposition, LLMs can be guided to solve tasks by eliciting intermediate logical stages, thereby offering both better accuracy and a degree of explainability [58], [59]. In our implementation, the model is guided through a systematic reasoning workflow, illustrated in Figure 15. It incorporates the key parts below:

1. **Deconstruct:** Break the user request into core components such as filters, aggregations, and desired outputs.

2. **Consult:** Leverage retrieved domain knowledge to clarify ambiguous terms and contextualize the question.

3. **Locate:** Map natural language phrases precisely to database schema elements (tables, columns) using metadata, avoiding hallucinated references.

4. **Formulate:** Compose the SQL query with strict syntax adherence, including fully qualified names, constraints, type-aware formatting, and proper string escaping.

5. **Review:** Self-verify the logical alignment between the query and the original question.

This step-wise approach simulates the staged problem-solving of a human expert, allowing the agent to decompose complex tasks, fully utilize the provided context, and produce more reliable, executable code.



Step-by-Step Reasoning Strategy:

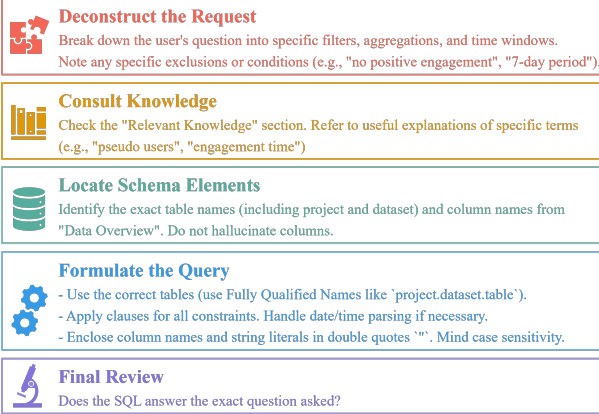


Fig. 15: Stage-aware Reasoning

C. Agent Prompt Design

In this section, we detail the specific prompt designs for the three core agents in our framework. We present representative prompts with clear hierarchical structures and modular components to ensure reproducibility and transparency.

1) **Generator Agent:** The Generator Agent is responsible for creating initial code based on user requests, database schemas, and contextual knowledge. It employs comprehensive reasoning strategies to ensure accurate and executable code generation while maintaining strict schema alignment.

a) **Core Mechanism:** The agent synthesizes code through a structured five-step reasoning process that decomposes user requests, consults knowledge bases, locates schema elements, formulates queries, and performs final validation. The process emphasizes the use of fully qualified table names and proper column referencing to prevent common errors in BigQuery environments.

Prompt for Generator Agent

Role: You are an excellent SQL expert.

Task: Write a SQL query to answer the user's request based on the provided database schema.

Contextual Inputs:

- User request: {query}
- Data Overview: {data_context}
- Relevant knowledge: {context}

Staged Reasoning Strategy:

- 1) **Deconstruct the Request:** Break down the user's question into specific filters, aggregations, and time windows. Note any specific exclusions or conditions (e.g., "no positive engagement", "7-day period").
- 2) **Consult Knowledge:** Check the "Relevant Knowledge" section. Refer to useful explanations of specific terms (e.g., "pseudo users", "engagement time").
- 3) **Locate Schema Elements:** Identify the exact table names (including project and dataset) and column names from "Data Overview". Do not hallucinate columns.
- 4) **Formulate the Query:**
 - Use the correct tables.
 - Apply clauses for all constraints.
 - Handle date/time parsing if necessary.
- 5) **Final Review:** Does the SQL answer the exact question asked?

Output Requirements:

- 1) First, provide a short Analysis of the question.
- 2) Then, output the SQL inside a Markdown code block.

Output Format:

Analysis: [Your brief thought process]

```
```\sql
SELECT (your SQL clause)
FROM {fqn}
(your SQL clause)
```\
```

(Replace with the actual table and column names using Fully Qualified Name)

b) **Error Handling and Safety Features:** The implementation includes robust error-handling mechanisms:

- **Content Filtering:** Automatic retry logic when API content filters are triggered, omitting external knowledge contexts while preserving core functionality
- **Context Truncation:** Intelligent truncation of lengthy data contexts (>350,000 characters) with

clear indication of truncation to maintain prompt size limits

- **FQN Resolution:** Automated extraction of fully qualified table names from available schema information

2) *Critic Agent:* The Critic Agent adopts the **Delta-guided Memory Management** mechanism, serving as a meta-cognitive component, which monitors the debugging life-cycle, evaluates correction effectiveness, and provides strategic guidance for overcoming persistent challenges. It functions as a quality assurance layer that prevents debugging loops and promotes systematic problem-solving.

a) *Reflection and Progress Assessment:* The agent implements a comparative analysis framework that evaluates whether previous fixes successfully resolved errors or created new issues. Tracking the evolution of errors and code changes, it provides actionable insights for navigating complex debugging scenarios.

Prompt for Critic Agent

Role: You are an expert Reviewer.

Goal: Check the debugging progress and provide actionable suggestions for the next step.

Reflection Context:

- Previous Error History: {history}
- Last Encountered Error: {last_error}
- Last SQL Code: {last_code}
- Current Error: {current_error}
- Current SQL Code: {current_code}

Reflection Process (Chain of Thought):

- 1) **Compare Errors:** Did the "Last Encountered Error" disappear in the "Current Error"?
 - If YES: The previous fix was successful (or at least moved past that bug).
 - If NO (Same Error): The previous fix FAILED. We are stuck in a loop.
- 2) **Formulate Advice:**
 - If progressing: Summarize the error. Compare the last SQL code with the current SQL code, and describe the solution to it.
 - If stuck: Suggest a different approach. You can also refer to the previous error history for some inspiration.

Output Format:

Current Code [{current_code}]. Current Error[{current_error}].
Previous fix [Worked/Failed]. Suggestion: [Action].

b) *Strategic Decision Support:* The Critic Agent provides binary classification of fix effectiveness

("Worked"/"Failed") coupled with concrete action suggestions. This clear directive format enables the debugging system to escape local optima and explore alternative correction strategies when standard approaches prove ineffective.

3) *Reviser Agent:* The Reviser Agent specializes in diagnosing and fixing execution errors through systematic debugging strategies. It operates as a corrective layer that analyzes error messages, historical error patterns, and reference hints to produce corrected code.

a) *Debugging Methodology:* The agent employs a four-phase debugging strategy that classifies errors, locates fault points, formulates targeted fixes, and ensures logical consistency. This structured approach prevents error recurrence and promotes learning from historical debugging patterns.

Prompt for Reviser Agent

Role: You are an excellent SQL expert.

Task: Correct the SQL based on the error message after the previous SQL query failed to execute.

Debugging Context:

- User request: {query}
- Latest executed SQL: {SQL}
- Error: {error}
- Error history: {error_history}
- Reference / Hints: {reference}

Step-by-Step Debugging Strategy (Chain of Thought):

- 1) **Analyze the Error:** Read the error message carefully. Is it a syntax error, a schema mismatch, or a logic issue?
- 2) **Locate the Fault:** Identify the exact line or clause in the SQL that caused the error.
 - If it's a syntax error, check the command and function usage.
 - If it's a schema mismatch, check the correct FQN for tables and correct case-sensitive columns.
 - If it's a logic issue, review the user request constraints (e.g., date ranges, filters) carefully.
- 3) **Formulate a Fix:** Make use of your knowledge and the reference/hints to fix the issues.
- 4) **Final Refinement:** Ensure the rest of the query (filters, logic) remains consistent with the User Request.
 - For syntax/schema: applying the correct function or name.
 - For logic: consider rewriting the CTEs or subqueries to align with the correct logic.

Output Requirements:

- 1) First, provide a short Analysis of why the error happened and how you will fix it.
- 2) Then, output the Corrected SQL inside a Mark-down code block.

Output Format:

Analysis: [Your brief debugging thought process]

```
``sql
SELECT (your SQL clause)
FROM {fqn}
(your SQL clause)
``
(Replace with the actual table and column names
using FQN)
```

b) Historical Error Learning: The Reviser Agent maintains and utilizes an error history context offered by the Critic Agent to identify recurring error patterns and prevent previously encountered issues. This historical awareness enables more intelligent debugging that evolves based on accumulated correction experiences.

4) System Integration and Workflow: The three agents operate in a coordinated pipeline:

- 1) **Generator Agent** produces initial SQL queries with comprehensive reasoning
- 2) **Reviser Agent** intervenes when execution errors occur, applying systematic debugging
- 3) **Critic Agent** monitors the debugging trajectory, providing meta-level guidance

This hierarchical structure ensures that SQL generation benefits from both immediate error correction and long-term debugging strategy optimization. The prompt designs emphasize consistency in output formatting, use of fully qualified names, and adherence to BigQuery-specific syntax conventions, creating a robust framework for automated SQL query generation and refinement.

D. Evaluation Metric Details

Let Y_n and $\hat{Y}_n$ denote the ground-truth and predicted SQL queries for the n -th example, respectively, and let V_n and $\hat{V}_n$ denote their corresponding execution results. Let N denote the number of examples in the evaluation set.

a) Execution Accuracy (EX): EX measures the proportion of examples for which the predicted SQL query returns results equivalent to those of the ground-truth query [60]. For BIRD and Spider 2.0-Lite, it is computed as [43], [44]:

$$EX = \frac{1}{N} \sum_{n=1}^N \mathbb{1}(V_n, \hat{V}_n),$$

where $\mathbb{1}(\cdot, \cdot)$ is an indicator function:

$$\mathbb{1}(V, \hat{V}) = \begin{cases} 1, & V = \hat{V}, \\ 0, & \text{otherwise.} \end{cases}$$

Here, equality is determined by the result-comparison rules of the corresponding benchmark. Predictions that fail to execute receive a score of zero.

b) Valid Efficiency Score (VES): VES evaluates execution efficiency while accounting for execution correctness. Only predictions whose execution results match the ground-truth results contribute to the score [44]:

$$VES = \frac{1}{N} \sum_{n=1}^N \mathbb{1}(V_n, \hat{V}_n) \rho_n,$$

$$\rho_n = \sqrt{\frac{T(Y_n)}{T(\hat{Y}_n)}}.$$

Here, $T(\cdot)$ denotes the execution time of a SQL query measured under the same evaluation environment, and ρ_n represents the relative execution efficiency of the predicted query. Incorrect or non-executable predictions contribute zero to VES.

c) Soft-F1: Soft-F1 measures partial agreement between the predicted and ground-truth execution results at the cell level. Let $\mathcal{C}(V_n)$ and $\mathcal{C}(\hat{V}_n)$ denote the sets of cell values in the ground-truth and predicted results, respectively. When both result sets are nonempty, the instance-level precision P_n and recall R_n are defined as:

$$P_n = \frac{|\mathcal{C}(V_n) \cap \mathcal{C}(\hat{V}_n)|}{|\mathcal{C}(\hat{V}_n)|},$$

$$R_n = \frac{|\mathcal{C}(V_n) \cap \mathcal{C}(\hat{V}_n)|}{|\mathcal{C}(V_n)|}.$$

Here, $|\cdot|$ denotes set cardinality. When both result sets are empty, we set $P_n = R_n = 1$; when exactly one is empty, we set $P_n = R_n = 0$. The instance-level Soft-F1 score is:

$$F_n = \begin{cases} \frac{2P_n R_n}{P_n + R_n}, & P_n + R_n > 0, \\ 0, & \text{otherwise.} \end{cases}$$

The overall score is the average across all evaluation examples:

$$\text{Soft-F1} = \frac{1}{N} \sum_{n=1}^N F_n.$$

E. Quantitative Application Case Supplements

1) Similar Challenges: In the field of quantitative investment and financial computing, LLMs are confronted with similar challenges mentioned in Section I:

Financial Domain Semantics: General-purpose LLMs typically lack deep semantic understanding of

specialized operators and functions in alpha factor construction (e.g. `delta`, `delay`, `adv{d}`, `industry neutralization`) [61]. Also, financial data are usually stored as large-scale tabular structures (CSV, DataFrames) with thousands of rows (e.g., daily trading records), which is hard for existing LLMs to handle [62], [63]. (Challenge 1)

Unreliable Numerical Operations: Factor calculations usually involve *rolling windows*, *cross-sectional rankings*, and *multi-level nesting*, where LLMs tend to lose intermediate states [62] and make computational errors. A snippet of 000006.SZ trading data is shown in Table XI.(Challenge 2)

TABLE XI: Sample of Stock Trading Data (000006.SZ)

Date	Open	High	Low	Close	...	Industry
2009-01-05	55.09	56.84	54.76	56.73	...	Real Estate
2009-01-06	56.73	59.03	56.18	59.03	...	Real Estate
⋮	⋮	⋮	⋮	⋮	⋮	⋮
2025-08-15	285.33	294.87	283.35	291.29	...	Real Estate

The Quantitative Analysis Agent system is designed to generate, execute, and debug Python scripts for quantitative financial analysis tasks. The system consists of three interconnected agents: **Generator Agent** for initial code generation, **Reviser Agent** for error correction, and **Critic Agent** for debugging progress monitoring.

2) *Domain Knowledge Injection:* The system incorporates extensive domain knowledge for quantitative finance calculations, providing definitions for financial metrics and mathematical operators.

Quantitative Finance Domain Knowledge Context

Input data definitions:

- `returns` = daily close-to-close returns
- `open`, `close`, `high`, `low`, `volume` = standard definitions for daily price and volume data
- `vwap` = daily volume-weighted average price
- `cap` = market cap
- `adv{d}` = average daily dollar volume for the past `d` days
- `IndClass` = a generic placeholder for a binary industry classification such as GICS, BICS, NAICS, SIC, etc.

Functions and operators: (Below, "{ }" stands for a placeholder)

- `abs(x)`, `log(x)`, `sign(x)` = standard definitions; same for the operators "+", "-", "*", "/", ">", "<", "==", "||", "x ? y : z"
- `rank(x)` = cross-sectional rank
- `delay(x, d)` = value of `x` `d` days ago

- `correlation(x, y, d)` = time-series correlation of `x` and `y` for the past `d` days
- `covariance(x, y, d)` = time-series covariance of `x` and `y` for the past `d` days
- `scale(x, a)` = rescaled `x` such that `sum(abs(x)) = a` (the default is `a = 1`)
- `delta(x, d)` = today's value of `x` minus the value of `x` `d` days ago
- `signedpower(x, a)` = x^a
- `decay_linear(x, d)` = weighted moving average over the past `d` days with linearly decaying weights `d, d - 1, ..., 1` (rescaled to sum up to 1)
- `indneutralize(x, g)` = `x` cross-sectionally neutralized against groups `g` (subindustries, industries, sectors, etc.)
- `ts_{O}(x, d)` = operator `O` applied across the time series for the past `d` days
- `ts_min(x, d)` = time-series min over the past `d` days
- `ts_max(x, d)` = time-series max over the past `d` days
- `ts_argmax(x, d)` = which day `ts_max(x, d)` occurred on
- `ts_argmin(x, d)` = which day `ts_min(x, d)` occurred on
- `ts_rank(x, d)` = time-series rank in the past `d` days
- `min(x, d)` = `ts_min(x, d)`
- `max(x, d)` = `ts_max(x, d)`
- `sum(x, d)` = time-series sum over the past `d` days
- `product(x, d)` = time-series product over the past `d` days
- `stddev(x, d)` = moving time-series standard deviation over the past `d` days

3) *Synthesis Pipeline:* The whole program generation process is shown in the Figure 10. The details are described as follows.

a) Initial Code Generation (Generator Agent):

The Generator Agent is responsible for creating initial Python scripts for quantitative analysis tasks based on user requests, data schemas, and domain knowledge. It employs structured implementation strategies to ensure accurate and efficient code generation.

Initialization Prompt: The core mechanism involves guiding the LLM through a four-step implementation strategy focusing on vectorized operations and proper data handling.

Prompt for Generator Agent (Abbreviated)

Role: You are an expert Quantitative Developer and Data Scientist.

Task: Write a Python script to perform quantitative analysis and calculation based on the user's request and provided CSV data.

Context:

- User request: {query}
- Data Overview (File paths & Columns): {data_context}
- Relevant Domain Knowledge (Formulas/Definitions): {context}

.....

b) *Code Error Correction (Reviser Agent)*: The Reviser Agent specializes in diagnosing and fixing Python execution errors through systematic debugging strategies. It operates in two modes: error correction mode and verification mode.

Error Correction Prompt: When the code fails to execute, the agent analyzes error messages and applies targeted fixes.

Prompt for Reviser Agent - Error Correction (Abbreviated)

Role: You are an expert Python Quantitative Developer.

Task: Fix the code based on the error message from previous execution.

.....

Verification Prompt: When the code executes successfully, the agent verifies if the logic and results match the user's request.

Prompt for Reviser Agent - Verification (Abbreviated)

Role: You are an expert Quantitative Reviewer.

Task: Verify if the logic and results match the User's Request.

c) *Delta-guided Memory (Critic Agent)*: The Critic Agent serves as a meta-cognitive component that monitors the debugging life-cycle, evaluates correction effectiveness, and provides strategic guidance.

Reflection Prompt: The agent analyzes debugging progress and suggests next steps based on error patterns.

Prompt for Critic Agent (Abbreviated)

Role: You are an expert Reviewer.

Goal: Check debugging progress and provide actionable suggestions.

4) *Program Pipeline for Factor computing*: The generated code for factor#36 (in WorldQuant's 101 Alphas [52]) calculation is implemented as follows.

Code Structure Summary:

- 1) **Data Loading** (lines 1–19): Load multiple CSV files and merge into a single DataFrame
- 2) **Preprocessing** (lines 22–33): Calculate daily returns, VWAP, and 20-day average dollar volume
- 3) **Time Series Ranking Function** (lines 36–40): Define the Ts_Rank helper function
- 4) **Alpha Factor Calculation** (lines 43–82): Compute the 5 Alpha factor components
- 5) **Result Aggregation** (lines 85–94): Combine components and convert to pivot table format

Key Calculations:

- **ts_rank**: Rolling time series ranking function
- **groupby().transform()**: Group-wise rolling calculations
- **pct=True**: Percentage ranking computation
- **rolling().corr()**: Rolling correlation coefficient calculation

```
1 import os
2 import pandas as pd
3 import numpy as np
4
5 # 1. Load all stock data
6 data_dir = "./stock_data"
7 all_files = [f for f in
8               os.listdir(data_dir) if
9               f.endswith('.csv')]
10
11 dfs = []
12 for file in all_files:
13     code = file.replace('.csv', '')
14     df = pd.read_csv(os.path.join(data_dir, file), dtype={'date': str})
15     df['code'] = code
16     dfs.append(df)
17 df = pd.concat(dfs, ignore_index=True)
18
19 # Convert date to datetime and set as index for easier manipulation
20 df['date'] = pd.to_datetime(df['date'])
21 df.sort_values(['code', 'date'], inplace=True)
22
23 # 2. Preprocessing: ensure numeric columns are numeric (they already are per schema)
24 # Compute returns: close-to-close daily returns
25 df['returns'] = df.groupby('code')['close'].pct_change()
26
27 # Compute vwap: volume-weighted average price (using amount and volume)
28 df['vwap'] = df['amount'] / df['volume'].replace(0, np.nan)
```

```

29
30 # Compute adv20: 20-day average dollar
    volume (amount)
31 df['adv20'] =
    df.groupby('code')['amount']
32 .transform( lambda x: x.rolling(20,
    min_periods=15).mean())
33
34 # 3. Calculate components of the alpha
    formula
35 # Helper function for time-series rank
    (ts_Rank)
36 def ts_rank(series, window):
37     # Rolling time-series rank: rank
        over the window, scaled to [0,1]
38     return series.rolling(window,
        min_periods=window//2).apply(
39         lambda x:
40             pd.Series(x).rank(pct=True)
        .iloc[-1], raw=False
41     )
42
43 # Component 1: 2.21 *
    rank(correlation((close - open),
        delay(volume, 1), 15))
44 df['close_open'] = df['close'] -
    df['open']
45 df['volume_lag1'] = df.groupby('code')
    ['volume'].shift(1)
46 # Rolling correlation over 15 days
47 df['corr_co'] =
    df.groupby('code').apply(lambda x:
48 x['close_open'].rolling(15, min_periods
    =10).corr(x['volume_lag1'])
49 ).reset_index(level=0, drop=True)
50 # Cross-sectional rank of correlation
51 df['rank_corr'] = df.groupby('date')
    ['corr_co'].rank(pct=True)
52 comp1 = 2.21 * df['rank_corr']
53
54 # Component 2: 0.7 * rank((open -
    close))
55 df['open_close'] = df['open'] -
    df['close']
56 df['rank_oc'] =
    df.groupby('date')['open_close'].
57 rank(pct=True)
58 comp2 = 0.7 * df['rank_oc']
59
60 # Component 3: 0.73 *
    rank(ts_Rank(delay((-1 * returns),
        6), 5))
61 df['neg_returns'] = -1 * df['returns']
62 df['neg_returns_lag6'] = df.groupby(
    'code')['neg_returns'].shift(6)
63 df['ts_rank_neg'] = df.groupby(
    'code')['neg_returns_lag6'].
64 transform(lambda x: ts_rank(x, 5)
65 )
66 df['rank_ts_rank'] =
    df.groupby('date')['ts_rank_neg'].
67 rank(pct=True)
68 comp3 = 0.73 * df['rank_ts_rank']
69
70

```

```

71 # Component 4:
    rank(abs(correlation(vwap, adv20,
        6)))
72 df['corr_vwap_adv20'] =
    df.groupby('code').apply(
73     lambda x: x['vwap'].rolling(6,
        min_periods=4).corr(x['adv20'])
74 ).reset_index(level=0, drop=True)
75 df['abs_corr'] =
    df['corr_vwap_adv20'].abs()
76 df['rank_abs_corr'] =
    df.groupby('date')['abs_corr'].rank(
77 pct=True)
78 comp4 = df['rank_abs_corr']
79
80 # Component 5: 0.6 *
    rank(((sum(close, 200) / 200) -
        open) * (close - open))
81 df['sum_close_200'] =
    df.groupby('code')['close'].
82 transform(
83     lambda x: x.rolling(200,
        min_periods=150).sum()
84 )
85 df['sma200'] = df['sum_close_200'] /
    200
86 df['sma200_open'] = df['sma200'] -
    df['open']
87 df['close_open2'] = df['close'] -
    df['open']
88 df['prod'] = df['sma200_open'] *
    df['close_open2']
89 df['rank_prod'] =
    df.groupby('date')['prod'].
90 rank(pct=True)
91 comp5 = 0.6 * df['rank_prod']
92
93 # 4. Combine all components
94 df['alpha'] = comp1 + comp2 + comp3 +
    comp4 + comp5
95
96 # 5. Pivot to desired format:
    index=date, columns=code,
    values=alpha
97 alpha_pivot =
    df.pivot_table(index='date',
        columns='code', values='alpha')
98
99 # 6. Output
100 print("Alpha factor pivot table
    shape:", alpha_pivot.shape)
101 print("\nFirst 5 rows and 5 columns:")
102 print(alpha_pivot.iloc[:5, :5])
103
104 alpha_pivot.to_csv("results.csv")

```