CoA-Cypher: Chain-of-Agents for Text-to-Cypher Generation

Anonymous ACL submission

Abstract

Text-to-Cypher generation converts natural-language questions into executable Cypher queries over property graph databases. Despite recent progress with Large Language Models (LLMs), this task remains challenging because of large and complex graph schemas and complex user queries. Existing zero-shot methods often generate invalid Cypher under large schemas or executable queries that follow the wrong graph path, limiting execution accuracy. We propose **CoA-Cypher**, a Chain-of-Agents framework that decomposes Text-to-Cypher generation into specialized stages. An **Analyzer** selects a question-specific sub-schema and routes structurally complex questions, a graph-aware **Decomposer** breaks complex questions into complementary partial Cypher forms, and a **Generator** produces the final Cypher query. To further improve correctness, a dual-agent feedback loop combines a syntactic **Critic** for execution-visible errors with a correctness-focused **Verifier** for executable-but-wrong queries. Experiments on CypherBench and Mind the Query show that CoA-Cypher achieves state-of-the-art execution accuracy, reaching **81.43%** EX on CypherBench and **82.96%** EX on Mind the Query, with consistent gains over existing works. Besides, an ablation study is also conducted to explore the behavior of the proposed method.

1 Introduction

The adoption of property graph databases, such as Neo4j, has accelerated the need for Natural Language Interfaces for Databases (NLIDB) (Miller, 2013). Translating natural language into Cypher queries (a standard declarative language for graphs) enables users to retrieve graph data without specialized query knowledge (Francis et al., 2018; Ganesan et al., 2024). While Large Language Models (LLMs) have demonstrated strong capabilities in relational Text-to-SQL tasks, Text-to-Cypher processing introduces distinct topological reasoning

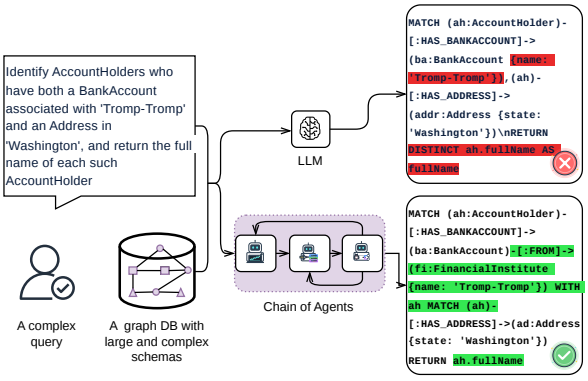

Figure 1: Unlike conventional methods (LLM) that rely on single-pass inference throughout query generation, CoA-Cypher allows the generation strategy to be adapted during execution. Multiple agents collaboratively assess intermediate outcomes and determine whether syntactic or correctness updates are required, enabling iterative refinement toward the final result.

challenges. Unlike standard SQL joins, property graphs encode semantics within multi-hop structures (Hornsteiner et al., 2024), which requires accurate reasoning over nodes, relationships, and traversal paths.

Standard zero-shot generation paradigms often suffer from degraded performance when applied to graph databases, primarily due to two fundamental challenges:

Challenge 1: Large and Complex Graph Schemas Real-world property graphs contain huge number of node labels, relationship types, and properties (Feng et al., 2025; Chauhan et al., 2025). Providing unpruned global schemas directly into the context window of an LLM induces severe information overload, resulting in schema-induced hallucinations and invalid syntax (Su et al., 2025).

Challenge 2: Highly Complex User Queries. Graph queries frequently involve intricate compositional logic, strict filtering constraints, and long-horizon multi-hop traversals (Francis et al., 2018;

Chauhan et al., 2025). Single-pass LLMs struggle with this depth of graph reasoning, ending up with generating queries that might execute without syntactic errors but retrieving data entirely misaligned with the user's intent (Chauhan et al., 2025).

To tackle these challenges directly, we propose **CoA-Cypher**, a Chain-of-Agents framework designed specifically for robust Text-to-Cypher generation. Instead of relying on a single monolithic inference step, CoA-Cypher decomposes the generation workflow across collaborative components tailored to the aforementioned challenges. To address Challenge 1, the **Analyzer** is deployed to prune the global graph schema into a compact, question-specific sub-schema, effectively eliminating context noise. To address Challenge 2, the **Analyzer** simultaneously computes query complexity signals (e.g., hop count) then dynamically delegates complex queries to the **Generator**, where a multi-strategy **Decomposer** synthesizes the query from multiple structural perspectives to prevent reasoning degradation. Finally, we improve accuracy through a dual-agent loop: a syntactic **Critic** resolves runtime errors, while a correctness-focused **Verifier** catches and corrects executable but intent-mismatched queries.

Our main contributions are summarized as follows:

- We propose **CoA-Cypher**, a multi-agent framework tailored for Text-to-Cypher translation. The framework combines schema pruning, complexity-aware analyzing, decomposition-based generation, and iterative refinement into a unified graph-query generation pipeline.

- We introduce a graph-aware multi-strategy decomposition mechanism that synthesizes Cypher queries from entity-centric, relation-centric, and constraint-centric perspectives. This process is further stabilized through a dual-agent feedback loop combining execution correction and intent-correctness verification.

- We conduct extensive experiments on two representative Text-to-Cypher benchmarks: CypherBench and Mind the Query. Results across multiple backbone LLMs demonstrate that CoA-Cypher consistently outperforms strong prompting and multi-agent baselines, especially on topologically complex graph queries.

2 Related Work

Text-to-Cypher. Graph databases (Miller, 2013) have emerged as an effective solution for modeling highly interconnected data across domains such as recommendation systems, knowledge graphs, and biomedical analytics. However, querying graph databases through Cypher (Francis et al., 2018; Ganesan et al., 2024; Liu et al., 2024) remains difficult for non-expert users because Cypher requires reasoning over graph topology and multi-hop traversals. Consequently, Natural Language to Cypher (NL2Cypher) generation has become an important research problem.

Compared with Text-to-SQL, Text-to-Cypher introduces additional structural complexity due to flexible traversal patterns, optional matches, variable-length paths, and path-aware aggregation (Hornsteiner et al., 2024). Existing approaches attempt to solve these challenges through schema-aware modeling and modular generation pipelines. For example, CoBGT (Tran et al., 2024) decomposes generation into relation prediction and Cypher decoding stages, while Cypher-RI (Su et al., 2025) introduces reinforcement-learning-based schema filtering.

While these methods improve schema grounding, they still rely heavily on static schema selection and local relation prediction, which often propagate irrelevant schema components and lead to hallucinated or incorrect traversals under large graph schemas. Moreover, despite improved training coverage from synthetic data generation strategies, current systems continue to exhibit weak answer accuracy on unseen schemas and complex graph structures (Tiwari et al., 2025).

To support benchmarking, several datasets have recently emerged for Text-to-Cypher evaluation, including Neo4j-Text2Cypher (Ozsoy et al., 2025), SyntheT2C (Zhong et al., 2025), AutoCypher (Tiwari et al., 2025), and execution-oriented benchmarks such as CypherBench (Feng et al., 2025) and Mind the Query (Chauhan et al., 2025).

Query Decomposition and Multi-Agent Collaboration. Single-pass generation remains fragile for complex structured query synthesis due to context limitations and reasoning degradation in long-horizon tasks. To resolve this issue, chain-of-thought and least-to-most prompting strategies (Tai

et al., 2023) have proved the effectiveness and reliability of decomposed reasoning, motivating staged systems including DIN-SQL (Pourreza and Rafiei, 2023), CQR-SQL (Xiao et al., 2022), and Struct-GPT (Jiang et al., 2023). Furthermore, recent multi-agent frameworks, including MAC-SQL (Wang et al., 2025) and AutoGen (Wu et al., 2024), further improve robustness through role-specialized collaboration and iterative correction.

Nevertheless, decomposition only pipelines still face important limitations of applying fixed decomposition strategies, introducing unnecessary reasoning overhead for simple queries. Furthermore, existing decomposition pipelines are primarily optimized for relational databases and fail to explicitly preserve graph topology or traversal logic during reasoning.

Self-Correction and Execution Feedback. Hallucination and structural inconsistency remain persistent problems in query generation. Prior works include iterative self-correction and execution-guided refinement have been widely adopted to improve structured query generation (Shinn et al., 2023; Zhu et al., 2026; Li et al., 2025, 2026). However, executable Cypher queries may still follow valid relationship paths while violating the user's intended reasoning logic. Existing execution-guided approaches cannot reliably detect these silent correctness failures because the query remains executable.

To address this limitation, **CoA-Cypher** introduces a dual-agent validation framework consisting of a syntactic critic and a correctness verifier. The critic repairs schema and execution errors, while the verifier explicitly checks alignment between the generated traversal logic and the original natural-language intent. This combination allows CoA-Cypher to correct both executable-but-wrong queries and structurally invalid generations.

3 Method

3.1 Problem Formulation

Given a natural-language question q and a graph database schema $\mathcal{S}$, Text-to-Cypher generates an executable Cypher query c that returns the answer intended by the user (Francis et al., 2018; Ozsoy et al., 2025; Feng et al., 2025). We represent the schema as $\mathcal{S} = (\mathcal{E}, \mathcal{R}, \mathcal{P})$, where $\mathcal{E}$ is the set of node labels, $\mathcal{R}$ is the set of directed relationship types, and $\mathcal{P}$ contains node and relationship properties. Each relationship $r \in \mathcal{R}$ is a typed directed edge (e_i, ℓ_r, e_j) between node labels $e_i, e_j \in \mathcal{E}$, where ℓ_r denotes the relationship-type label associated with r.

We formulate this generation process as a schema-grounded mapping:

$$c = f(q, \mathcal{S}), \tag{1}$$

where f maps the question intent and the graph schema into a Cypher query. The generated query should be schema-valid, executable, and aligned with q.

The main challenge is that correctness depends on both query syntax and graph topology. A valid-looking Cypher query may still be wrong if it uses an incorrect anchor node, edge direction, intermediate entity, or multi-hop traversal path. Thus, c must preserve the entities, constraints, aggregations, result modifiers, and traversal patterns expressed in q.

3.2 Overview

Figure 2 presents CoA-Cypher as a chain of specialized agents for Text-to-Cypher generation. Given a question q and schema $\mathcal{S}$, the **Analyzer** first grounds the question in a compact sub-schema and extracts complexity signals, such as filtering constraints, traversal hops, and aggregation requirements, for routing. A heuristic routing decision then determines whether the question can be generated directly or requires the **Decomposer** to split it into sub-questions and partial-Cypher pairs under three strategies. The query generated from the **Generator** is then executed to determine the feedback path. If execution fails, the **Critic** diagnoses syntax or schema errors and either sends patch feedback for query regeneration or requests schema re-selection. If execution succeeds, the **Verifier** checks whether the query still matches the question intent and triggers targeted regeneration when correctness drift is detected.

3.3 Analyzer

Analyzer prepares the input for the rest of the agent chain. It selects a question-specific sub-schema by keeping the labels, relationships, and properties needed for the input question. It also extracts analysis signals about constraints, traversal hops, aggregations, and result modifiers. This gives subsequent agents a smaller schema context and makes the main graph structure of the question explicit.

The Analyzer takes a user query and schemas as input. Its first output is a selected sub-schema

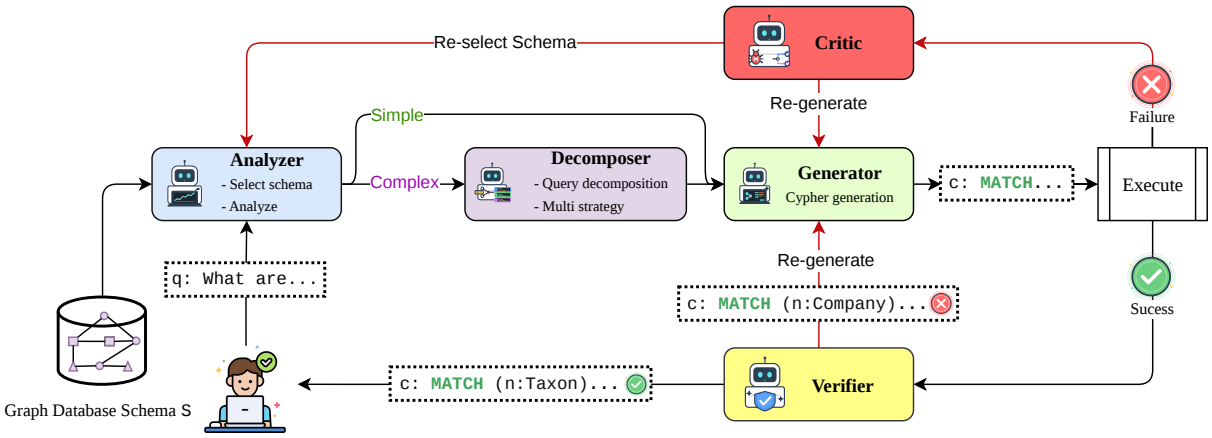

Figure 2: Overall architecture of CoA-Cypher. The Analyzer first selects a question-specific sub-schema and estimates structural complexity. Simple questions are sent directly to the Generator, while complex questions are routed through the Decomposer for multi-strategy planning. The generated query is executed and then refined by the Critic for execution failures and by the Verifier for correctness mismatches; both feedback agents can trigger targeted regeneration.

$\mathcal{S}_{\text{sub}} \subseteq \mathcal{S}$. This sub-schema keeps the labels, relationships, and properties that are needed for the question, and removes schema elements that are not relevant. The selected relationships are not treated as isolated edges; together, they should form a schema-valid traversal that connects the entities required by the question. Besides schema selection, the Analyzer also returns a short analysis of the question:

$$a = (\mathcal{F}, h, \mathcal{A}, \mathcal{M}), \tag{2}$$

where $\mathcal{F}$ is the set of filtering constraints, h is the estimated number of hops, i.e., the number of relationship traversals needed to connect the required entities, $\mathcal{A}$ is the set of aggregation functions such as `COUNT`, `AVG`, `SUM`, `MIN`, and `MAX`, and $\mathcal{M}$ contains result modifiers such as `DISTINCT`, `ORDER BY`, and `LIMIT`. These analysis signals make the main structure of the question explicit for later steps. For routing, we summarize them as complexity signals $\gamma = (n_{\text{const}}, n_{\text{hop}}, b_{\text{agg}})$:

- $n_{\text{const}} = |\mathcal{F}|$: the number of independent filtering constraints.
- $n_{\text{hop}} = h$: the estimated number of relationship traversals needed to connect the required entities.
- b_{agg}: a binary indicator of whether $\mathcal{A}$ is non-empty.

3.4 Multi-strategy Decomposer

Decomposer is activated only when the question requires additional structural planning. Before invoking it, CoA-Cypher applies a heuristic router to the Analyzer's output. The router uses the complexity signals γ to select a generation route $m \in \{\text{simple}, \text{complex}\}$. Formally, $\gamma = (n_{\text{const}}, n_{\text{hop}}, b_{\text{agg}})$, where n_{const} denotes the number of filtering constraints, n_{hop} denotes the estimated number of relationship hops, and $b_{\text{agg}} \in \{0, 1\}$ indicates whether aggregation is required. The routing decision is defined as:

$$m = \begin{cases} \text{complex}, & \text{if } \omega \\ \text{simple}, & \text{otherwise}, \end{cases} \tag{3}$$

where the complexity predicate ω is defined as $(n_{\text{const}} \geq \theta_{\text{const}}) \vee (n_{\text{hop}} \geq \theta_{\text{hop}}) \vee (b_{\text{agg}} = 1)$, with θ_{const} and θ_{hop} denoting the constraint and hop thresholds. If $m = \text{simple}$, the question bypasses the Decomposer and is sent directly to the Generator with $(q, \mathcal{S}_{\text{sub}})$. If $m = \text{complex}$, the Decomposer first produces a set of graph-aware strategies $\mathcal{D}$, which the Generator later uses to create candidate queries $\mathcal{C}_{\text{cand}}$.

For complex questions, Decomposer produces $n_{\text{strat}} = 3$ strategies, each consisting of sub-questions paired with corresponding sub-Cypher sketches. The first strategy is *entity-anchored*: it starts from the main entity mentioned in the question and expands outward through schema-valid relationships. The second is *relation/path-anchored*: it begins from the relationship structure needed to connect the required entities. The third is *constraint/aggregation-anchored*: it starts from filtering conditions, aggregation requirements, and output modifiers such as `DISTINCT`, `ORDER BY`, or `LIMIT`. The resulting decomposition set is denoted

Algorithm 1 Analyzer and Complexity-Aware Routing

Require: Question q, full schema $\mathcal{S}$, thresholds $\theta_{\text{const}}, \theta_{\text{hop}}$
Ensure: Sub-schema $\mathcal{S}_{\text{sub}}$, analysis signals a, complexity signals γ, route m

```
(S_sub, a) ← Analyzer(q, S)
Unpack a = (F, h, A, M)
γ ← (n_const, n_hop, b_agg) ← (|F|, h, |A| > 0)
if n_const ≥ θ_const or n_hop ≥ θ_hop or b_agg then
    m ← complex
else
    m ← simple
end if
return (S_sub, a, γ, m)
```

by $\mathcal{D} = \{d^{(1)}, \ldots, d^{(n_{\text{strat}})}\}$.

3.5 Generator

Generator synthesizes executable Cypher under the selected sub-schema. Its input depends on the route selected by the router. If $m =$ simple, CoA-Cypher treats the question as a direct mapping problem and invokes the Generator once with the original question q and selected sub-schema $\mathcal{S}_{\text{sub}}$:

$$c = \text{Generator}(q, \mathcal{S}_{\text{sub}}). \tag{4}$$

This route targets atomic lookups and lightly constrained one-hop questions, allowing CoA-Cypher to keep the reasoning path compact while grounding the query in the selected schema.

If $m =$ complex, the Generator is called once for each decomposition strategy to synthesize a complete candidate query, yielding a candidate set $\mathcal{C}_{\text{cand}} = \{c^{(1)}, \ldots, c^{(n_{\text{strat}})}\}$:

$$c^{(k)} = \text{Generator}(q, \mathcal{S}_{\text{sub}}, d^{(k)}), \tag{5}$$

where $d^{(k)}$ denotes the sub-question and sub-Cypher plan for the k-th decomposition strategy. CoA-Cypher then generates a single final query from $\mathcal{C}_{\text{cand}}$ before execution.

After generation, the query is executed against the graph database. The execution result determines whether the next agent is the Critic or the Verifier.

3.6 Feedback Agents

Critic and **Verifier** are two feedback agents with complementary roles. The Critic handles execution-visible failures, including syntax and schema-related errors raised during query execution, and can trigger query regeneration after producing patch feedback. The Verifier handles correctness mismatches between an executable Cypher query and the original natural-language question.

Algorithm 2 Complex-Route Decomposition and Generation

Require: Question q, selected sub-schema $\mathcal{S}_{\text{sub}}$, number of decomposition strategies n_{strat} (=3)
Ensure: Decomposition strategy set $\mathcal{D}$, candidate Cypher set $\mathcal{C}_{\text{cand}}$

```
D ← Decomposer(q, S_sub, n_strat)
Let D = {d^(k)}_{k=1}^{n_strat}, where each d^(k) contains sub-questions and sub-Cypher sketches.
C_cand ← ∅
for k = 1, ..., n_strat do
    c^(k) ← Generator(q, S_sub, d^(k))
    C_cand ← C_cand ∪ {c^(k)}
end for
return (C_cand, D)
```

Critic. When trial execution fails with an execution error ε, the Critic diagnoses whether the failure is primarily syntactic or schema-related and emits structured patch feedback:

$$f_{\text{critic}} = \text{Critic}(c, \varepsilon, q, \mathcal{S}_{\text{sub}}) \tag{6}$$

Concretely, the Critic audits the failed query against $\mathcal{S}_{\text{sub}}$ at the level of node labels, relationship types, and properties, then classifies the failure as `syntax`, `schema`, or `unknown`. Its structured output contains `status`, `error_type`, `feedback_target`, `schema_issues`, `patch_instructions`, and `regenerate`. The key routing variable is `feedback_target`: if the needed schema elements are already present but the query is incorrectly structured, feedback is sent to the Generator for regeneration; if the selected sub-schema is incomplete (e.g., missing label/relation/path), feedback is sent back to the Analyzer:

$$(\mathcal{S}'_{\text{sub}}, a') = \text{Analyzer}_{\text{fb}}(q, \mathcal{S}, \mathcal{S}_{\text{sub}}, c, f_{\text{critic}}) \tag{7}$$

The `patch_instructions` field provides minimal actionable edits for the next iteration, while `regenerate` controls whether the loop should invoke the Generator again or stop.

Verifier. A frequent silent-failure mode is that a query executes but does not satisfy the correctness requirements of the question. To address this, the **Verifier** performs answer-correctness validation by comparing the intent extracted from q with the result logic implied by c:

$$f_{\text{ver}} = \text{Verifier}(c, q, \mathcal{S}_{\text{sub}}) \tag{8}$$

Unlike the Critic, this verifier only drives Cypher-level revision and does not trigger schema re-selection. Both feedback agents operate directly

Algorithm 3 Dual-Agent Feedback Refinement

Require: Initial Cypher query c, question q
Full schema $\mathcal{S}$, sub-schema $\mathcal{S}_s$, decomposition context d, analysis a, max rounds $t_{\max}$
Ensure: Refined Cypher query c

```
for t = 1, ..., t_max do
    (ok, ε) ← Execute(c)
    if not ok then
        f ← Critic(q, S_s, c, ε)
        if not f.regenerate then
            break
        end if
        if f.feedback_target = analyzer then
            (S_s, a') ← Analyzer_fb(q, S, S_s, c, f)
            a ← a'
            c_new ← Generator(q, S_s, d, a)
        else
            c_new ← Generator_fb(c, f, q, S_s, d, a, ε)
        end if
    else
        f ← Verifier(q, S_s, c)
        if f.status ≠ needs_fix or not f.regenerate then
            break
        end if
        c_new ← Generator_fb(c, f, q, S_s, d, a, ε)
    end if
    if c_new = c then
        break
    end if
    c ← c_new
end for
return c
```

on $(q, \mathcal{S}_{\text{sub}}, c)$ and execution status. In implementation, the feedback loop does not re-run the Decomposer; it reuses the existing decomposition context d (empty in the simple branch, merged sub-questions in the complex branch). In either path, structured feedback constrains the next refinement step:

$$\begin{cases} c' = \text{Generator}_{\text{fb}}(c, f, q, \mathcal{S}_{\text{sub}}, d, a, \varepsilon), \\ f \in \{f_{\text{critic}}, f_{\text{ver}}\} \end{cases} \tag{9}$$

After the Generator outputs an initial final Cypher query, CoA-Cypher enters the feedback loop for iterative refinement as shown in Algorithm 3.

4 Experiments

4.1 Experimental Setup

Datasets. We evaluate CoA-Cypher on two complementary benchmarks designed to stress large, complex graph schemas and intricate query reasoning.

- **CypherBench** (Feng et al., 2025): a benchmark built from 11 Wikidata-derived property graphs containing 7.8M entities in total. Each graph exposes 4-8 entity types and 5-12 relation types together with a dense set of node and edge properties, producing large-scale, heterogeneous schemas that exceed the context budget of typical zero-shot prompting. The test split contains 2,348 questions across 7 domains, spanning multi-hop traversals, compositional constraints, and aggregations.
- **Mind the Query** (Chauhan et al., 2025): a benchmark whose test split comprises 7,059 NL-Cypher pairs drawn from 3 large, held-out knowledge graphs covering distinct domains: Healthcare Analytics, Women World Cup, and Bloom. Each NL-Cypher pair is verified through a multi-stage pipeline combining automated schema, runtime, and value validators with human-in-the-loop logical-correctness review, and is further stratified by query complexity into five tiers: **SR** (Simple Retrieval), **SA** (Simple Aggregation), **CR** (Complex Retrieval), **CA** (Complex Aggregation), and **EQ** (Evaluation Queries).

4.2 Main Results

Evaluation Metrics. Following prior works (Feng et al., 2025; Chauhan et al., 2025), we use the following metrics: **EX** (Execution Accuracy; primary metric), **PSJS** (Provenance Subgraph Jaccard Similarity; structure-aware overlap), and **Exec.** (Executable Percentage; syntax/schema validity).

Baselines. We compare against three baselines: **Direct** (single-pass generation), **CoT** (chain-of-thought reasoning before generation), and **MAC-SQL** (Wang et al., 2025), a representative multi-agent Text-to-SQL framework that we adapt to Text-to-Cypher by re-implementing its agent prompts and schema representations to target property graphs.

We evaluate all methods using four backbone LLMs. Three are open-source models: **GPT-OSS-120B**,[1] **Llama-3.3-70B-Instruct**,[2] and **Nemotron-3-Super-120B-A12B**.[3] We also include **MiniMax-M2.7**,[4] a strong closed-source model for code generation. The complete details of the implementation are provided in the Appendix B.

[1] https://huggingface.co/openai/gpt-oss-120b
[2] https://huggingface.co/meta-llama/Llama-3.3-70B-Instruct
[3] https://build.nvidia.com/nvidia/nemotron-3-super-120b-a12b
[4] https://www.minimax.io/models/text/m27

Method	CypherBench			Mind the Query		
	EX (%)	PSJS (%)	Exec. (%)	EX (%)	PSJS (%)	Exec. (%)
GPT-OSS-120B						
Direct	50.94	56.14	96.72	74.17	84.50	98.60
CoT	56.86	61.14	97.57	73.07	83.02	98.27
MAC-SQL	74.53	70.46	98.85	75.53	84.67	99.87
CoA-Cypher (Ours)	**81.43**	**76.38**	**99.96**	**82.96**	**89.00**	**99.99**
Llama-3.3-70B-Instruct						
Direct	35.69	45.66	85.31	65.76	77.51	88.10
CoT	39.74	49.50	82.71	63.39	75.48	86.66
MAC-SQL	53.53	56.58	92.55	66.24	76.47	93.02
CoA-Cypher (Ours)	**58.97**	**61.33**	**96.97**	**71.23**	**77.61**	**93.61**
Nemotron-3-Super-120B-A12B						
Direct	43.82	47.59	94.85	75.66	85.88	96.80
CoT	59.16	58.15	95.02	72.49	84.26	96.20
MAC-SQL	73.04	68.05	97.19	74.23	84.71	98.05
CoA-Cypher (Ours)	**76.28**	**72.66**	**99.32**	**80.80**	**89.08**	**99.69**
MiniMax-M2.7						
Direct	50.17	54.19	92.29	71.43	78.97	93.75
CoT	66.01	66.32	94.34	70.46	79.34	97.94
MAC-SQL	59.24	60.12	99.02	72.28	79.95	98.12
CoA-Cypher (Ours)	**70.74**	**68.76**	**99.28**	**80.72**	**86.53**	**99.46**

Table 1: Main results on CypherBench and Mind the Query across four LLMs. **Bold**: best per group; underline: second best per group.

Method	EX (%)	PSJS (%)	Exec. (%)
Gemini-1.5-Pro	39.95	57.70	86.03
GPT-4o	60.18	76.87	94.90
Claude-3.5-Sonnet	61.58	**80.85**	96.34
Cypher-RI	69.59	75.21	99.28
CoA-Cypher (Ours)	**81.43**	76.38	**99.96**

Table 2: Comparison with prior SOTA on CypherBench (EX, PSJS, Exec.). CoA-Cypher (GPT-OSS-120B) sets the new state-of-the-art on EX. Baseline results are from (Feng et al., 2025) and (Su et al., 2025). **Bold**: best; underline: second best.

Method	CA	CR	EQ	SA	SR	Total
Phi-4	26.23	50.63	81.07	67.53	80.60	62.01
Llama-4-Maverick	24.05	66.46	91.71	82.39	86.40	70.97
DeepSeek-V3	49.93	69.04	89.02	79.97	92.76	76.75
CoA-Cypher (Ours)	**55.49**	**70.86**	**98.82**	**89.55**	**95.35**	**82.96**

Table 3: Comparison with prior SOTA on Mind the Query (zero-shot EX per query category). CoA-Cypher (GPT-OSS-120B) sets the new state-of-the-art on all five categories and Total EX. Baseline results are from (Chauhan et al., 2025). **Bold**: best; underline: second best.

Comparison with Baselines. Table 1 presents the overall comparison across backbone LLMs and Text-to-Cypher benchmarks. CoA-Cypher achieves the best EX, PSJS, and Exec. scores in every model benchmark group.

Compared with Direct, CoA-Cypher consistently improves EX across all model families; under GPT-OSS-120B, for example, EX rises from 50.94% to **81.43%** on CypherBench and from 74.17% to **82.96%** on Mind the Query. Notably, several baselines achieve high Exec. scores while still underperforming on EX and PSJS, confirming that high executability does not guarantee answer correctness and motivating the Verifier component.

CoA-Cypher also improves consistently over MAC-SQL. Under GPT-OSS-120B, EX increases by **+6.9%** on CypherBench and **+7.4%** on Mind the Query. This suggests that graph query generation requires specialized handling beyond conventional Text-to-SQL decomposition pipelines.

Comparison with prior SOTA. Tables 2 and 3 compare CoA-Cypher with the strongest previously reported systems on the two benchmarks. On the primary metric, CoA-Cypher establishes a new state-of-the-art result on both datasets.

Settings	CypherBench			Mind the Query		
	EX (%)	PSJS (%)	Exec. (%)	EX (%)	PSJS (%)	Exec. (%)
S1: Generator only	50.94	56.14	96.72	74.17	84.50	98.60
S2: S1 + Analyzer	74.45	75.26	98.81	78.41	85.37	99.16
S3: S2 + Decomposer	76.92	75.61	98.89	82.18	88.61	98.74
S4: S3 + Critic	79.98	75.83	**99.96**	82.24	88.52	**99.99**
S5: S4 + Verifier [Full]	**81.43**	**76.38**	**99.96**	**82.96**	**89.00**	**99.99**

Table 4: Ablation results on CypherBench and Mind the Query. Each row incrementally adds one agent over the previous setting. S1 = Direct baseline; S5 = CoA-Cypher. **Bold**: best; underline: second best.

On CypherBench, CoA-Cypher reaches **81.43%** EX, outperforming Cypher-RI the strongest prior method by **+11.8%** and substantially exceeding GPT-4o (60.18%) and Claude-3.5-Sonnet (61.58%) reported by (Feng et al., 2025). Notably, these gains are achieved in a fully zero-shot prompting setting, without any task-specific training.

On Mind the Query, Table 3 evaluates per-category execution accuracy across the five complexity tiers (SR, SA, CR, CA, EQ; see Section 4.1). CoA-Cypher achieves **82.96%** Total EX, outperforming the best previously reported zero-shot baseline, DeepSeek-V3 (76.75%), by **+6.2%** and setting the best score in every category. The gains are largest on the hardest tiers **CA** (**+5.6%** over DeepSeek-V3) and **EQ** (**+7.1%** over Llama-4-Maverick) which are precisely the categories that demand long-horizon multi-hop reasoning and are most exposed to silent correctness errors.

4.3 Ablation Study

We evaluate five incrementally constructed configurations to isolate the contribution of each agent, all using GPT-OSS-120B (Table 4): **S1** uses only the Generator with the full schema (equivalent to Direct); **S2** adds the Analyzer for schema pruning and complexity estimation; **S3** adds the Decomposer for multi-strategy complex-query generation; **S4** adds a Feedback Loop with the Critic to correct execution-visible failures; **S5** completes the feedback loop with a Verifier for answer-correctness checking (Full CoA-Cypher).

Each added component contributes positively to EX. The Analyzer (S1→S2) produces the largest single gain (**+23.5%** on CypherBench), confirming that schema pruning is the primary lever against complex graph schemas: collapsing the global schema into a compact, question-specific sub-schema directly suppresses context overload and hallucinated traversals (Su et al., 2025). The Decomposer (S2→S3) adds further gains, particularly on Mind the Query, by breaking down complex multi-hop queries into topology-aware subplans, preventing reasoning degradation across long traversal chains (Tai et al., 2023; Wang et al., 2025).

The Feedback Agents deliver substantial additional gains in EX, particularly on the harder benchmark. The Critic (S3→S4) corrects execution-visible failures and, when necessary, triggers sub-schema re-selection, lifting EX by **+3.06%** on CypherBench while driving Exec. to 99.96%. The Verifier (S4→S5) targets the complementary class of silent correctness failures: queries that execute but traverse incorrect entities or edge directions due to complex compositional constraints, adding a further **+1.45%** EX on CypherBench and **+0.72%** EX on Mind the Query. Appendix A presents a qualitative case study illustrating how decomposition prevents such silent failures on the Mind the Query Healthcare graph.

5 Conclusion

We presented CoA-Cypher, a Chain-of-Agents framework for Text-to-Cypher generation. The Analyzer prunes the global schema into a compact, question-specific sub-schema and routes each query based on its structural complexity. For complex queries, the Decomposer breaks the problem down into multiple graph-aware strategies, and the Generator synthesizes the final candidate. A dual-agent feedback loop combining a syntactic Critic and a correctness-focused Verifier then iteratively corrects both execution errors and silent correctness failures. Extensive evaluations on CypherBench and Mind the Query confirm that CoA-Cypher consistently achieves new state-of-the-art execution accuracy across multiple LLM backbones.

6 Limitations

While CoA-Cypher demonstrates strong performance across multiple benchmarks, several limitations remain. First, the multi-agent pipeline introduces additional computational overhead compared to single-pass generation approaches. Each query may invoke multiple LLM calls across the Analyzer, Decomposer, Generator, Critic, and Verifier, leading to increased latency and inference cost, particularly for queries routed to the complex branch. Furthermore, all components currently share the same backbone LLM without component-specific optimization, leaving potential performance gains from specialized or fine-tuned models unexplored. Future work could explore lightweight or distilled models for individual components to reduce inference overhead while maintaining generation quality.

Second, the prompts used across all components are designed to be model-agnostic and have not been tailored to individual backbone LLMs. Model-specific prompt optimization could potentially yield further improvements, particularly for open-weight models. Similarly, the complexity thresholds θ_{const} and θ_{hop} used by the Heuristic Router are fixed heuristics rather than learned parameters, which may limit adaptability to graph schemas with substantially different structural characteristics. Learning these thresholds from data, or replacing the heuristic router with a trained classifier, represents a promising direction for future work.

References

Vashu Chauhan, Shobhit Raj, Shashank Mujumdar, Avirup Saha, and Anannay Jain. 2025. Mind the query: A benchmark dataset towards text2cypher task. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, EMNLP 2025 - Industry Track, Suzhou, China, November 4-9, 2025*, pages 1890–1905. Association for Computational Linguistics.

Yanlin Feng, Simone Papicchio, and Sajjadur Rahman. 2025. CypherBench: Towards precise retrieval over full-scale modern knowledge graphs in the LLM era. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8934–8958, Vienna, Austria. Association for Computational Linguistics.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA. Association for Computing Machinery.

Balaji Ganesan, Sambit Ghosh, Nitin Gupta, Manish Kesarwani, Sameep Mehta, and Renuka Sindhgatta. 2024. Llm-powered graphql generator for data retrieval. In *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 8657–8660. International Joint Conferences on Artificial Intelligence Organization. Demo Track.

Markus Hornsteiner, Michael Kreussel, Christoph Steindl, Fabian Ebner, Philip Empl, and Stefan Schönig. 2024. Real-time text-to-cypher query generation with large language models for graph databases. *Future Internet*, 16(12).

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Xin Zhao, and Ji-Rong Wen. 2023. StructGPT: A general framework for large language model to reason over structured data. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 9237–9251, Singapore. Association for Computational Linguistics.

Haoran Li, Yifan Sun, Wenjie Zhao, and Dong Xu. 2025. Emlc: Error multi-level correction framework for text-to-sql parsing. *Information Processing & Management*.

Xiang Li, Jinguo You, Heng Li, Jun Peng, Xi Chen, Ziheng Guo, Kun Li, and Tianyi Xu. 2026. G²sql: guided & guarded text-to-sql generation with two-stage verification. *Expert Systems with Applications*, 311:131276.

Yang Liu, Xin Wang, Jiake Ge, Hui Wang, Dawei Xu, and Yongzhe Jia. 2024. Text to graph query using filter condition attributes. *Proceedings of the VLDB Endowment*.

Justin J. Miller. 2013. Graph database applications and concepts with neo4j.

Makbule Gulcin Ozsoy, Leila Messallem, Jon Besga, and Gianandrea Minneci. 2025. Text2cypher: Bridging natural language and graph databases. In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025 - Workshops, Abu Dhabi, UAE, January 19-24, 2025*, pages 100–108. Association for Computational Linguistics.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Hanchen Su, Xuyuan Li, Yan Zhou, zhuoyi lu, Ziwei Chai, Haozheng Wang, Chen Zhang, and YANG YANG. 2025. Cypher-ri: Reinforcement learning for integrating schema selection into cypher generation. In *Advances in Neural Information Processing Systems*, volume 38, pages 100204–100225. Curran Associates, Inc.

Chang-Yu Tai, Ziru Chen, Tianshu Zhang, Xiang Deng, and Huan Sun. 2023. Exploring chain of thought style prompting for text-to-SQL. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5376–5393, Singapore. Association for Computational Linguistics.

Aman Tiwari, Shiva Krishna Reddy Malay, Vikas Yadav, Masoud Hashemi, and Sathwik Tejaswi Madhusudhan. 2025. Auto-cypher: Improving LLMs on cypher generation via LLM-supervised generation-verification framework. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 2: Short Papers)*, pages 623–640, Albuquerque, New Mexico. Association for Computational Linguistics.

Quoc-Bao-Huy Tran, Aagha Abdul Waheed, and Sun-Tae Chung. 2024. Robust text-to-cypher using combination of bert, graphsage, and transformer (cobgt) model. *Applied Sciences*, 14(17):7881.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2025. MAC-SQL: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 540–557. Association for Computational Linguistics.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*.

Dongling Xiao, LinZheng Chai, Qian-Wen Zhang, Zhao Yan, Zhoujun Li, and Yunbo Cao. 2022. CQR-SQL: Conversational question reformulation enhanced context-dependent text-to-SQL parsers. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, pages 2055–2068, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zijie Zhong, Linqing Zhong, Zhaoze Sun, Qingyun Jin, Zengchang Qin, and Xiaofan Zhang. 2025. Synthet2c: Generating synthetic data for fine-tuning large language models on the text2cypher task. In *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 672–692. Association for Computational Linguistics.

Chunlin Zhu, Yuming Lin, Yaojun Cai, and You Li. 2026. Scot2s: Self-correcting text-to-sql parsing by leveraging llms. *Comput. Speech Lang.*, 95:101865.

A Qualitative Case Study

We present a representative example drawn from the *Mind the Query* dataset (Healthcare graph) to illustrate how CoA-Cypher's multi-strategy decomposition leads to correct query generation where single-pass baselines fail.

Example Query

Natural Language Question

For cases reported by `'Consumer'`, what are the top 3 most frequent `route` values used in the `IS_PRIMARY_SUSPECT` relationships, and how many cases are associated with each route?

This question is structurally complex for three reasons: (i) it requires a two-hop traversal (`Case` → `ReportSource` and `Case` → `Drug`), (ii) the filter on `'Consumer'` applies to a node property of `ReportSource` reached via the `REPORTED_BY` relationship - not a property of `Case` itself, and (iii) the target property `route` resides on the `IS_PRIMARY_SUSPECT` relationship, not on any node.

Relevant Sub-Schema

Nodes:

`Case`	`reporterOccupation`, `gender`, `age`, ...
`ReportSource`	`name`, `code`
`Drug`	`name`, `primarySubstance`

Relationships:

`(Case)-[:REPORTED_BY]->(ReportSource)`	-
`(Case)-[:IS_PRIMARY_SUSPECT]->(Drug)`	`route`, `indication`, `doseAmount`, ...

Key distinction: `Consumer` is a value of `ReportSource.name`, **not** `Case.reporterOccupation`.

Gold Standard Query

```
MATCH (c:Case)-[:REPORTED_BY]->(rs:ReportSource {name: 'Consumer'}),
      (c)-[r:IS_PRIMARY_SUSPECT]->(d:Drug)
RETURN r.route AS route, COUNT(c) AS caseCount
ORDER BY caseCount DESC
LIMIT 3
```

Comparison of Generated Queries

Direct

```
MATCH (c:Case)-[r:IS_PRIMARY_SUSPECT]
      ->(:Drug)
WHERE c.reporterOccupation = 'Consumer'
  AND exists(r.route)
RETURN r.route AS route,
       count(DISTINCT c) AS caseCount
ORDER BY caseCount DESC
LIMIT 3
```

CoT

```
MATCH (c:Case)-[r:IS_PRIMARY_SUSPECT]
      ->(:Drug)
WHERE c.reporterOccupation = 'Consumer'
  AND r.route IS NOT NULL
WITH r.route AS route,
     COUNT(DISTINCT c) AS caseCount
RETURN route, caseCount
ORDER BY caseCount DESC, route
LIMIT 3
```

MAC-SQL

```
MATCH (c:Case)
WHERE c.reporterOccupation = 'Consumer'
WITH c
MATCH (c)-[r:IS_PRIMARY_SUSPECT]
      ->(:Drug)
WITH r.route AS route,
     COUNT(DISTINCT c) AS caseCount
ORDER BY caseCount DESC
LIMIT 3
RETURN route, caseCount
```

CoA-Cypher (Ours)

```
MATCH (c:Case)-[:REPORTED_BY]->
  (rs:ReportSource {name: 'Consumer'})
MATCH (c)-[r:IS_PRIMARY_SUSPECT]
      ->(:Drug)
WITH r.route AS route,
     count(DISTINCT c) AS caseCount
ORDER BY caseCount DESC
LIMIT 3
RETURN route, caseCount
```

Error Analysis

All three baselines produce executable but incorrect queries. Each method filters on `c.reporterOccupation = 'Consumer'`, conflating two structurally distinct schema elements: the `Case` node property `reporterOccupation` and the `ReportSource` node property `name`, which is reachable only via the `REPORTED_BY` relationship. Under a dense schema with multiple `Case` properties, we hypothesize that single-pass generation tends to anchor filters directly on the nearest node rather than traversing to the correct neighboring entity.

How Decomposition Leads to the Correct Query

The Heuristic Router classifies this question as *complex* ($n_{\text{hop}} = 2$, $b_{\text{agg}} = 1$), activating the Decomposer. Among the three generated strategies, Strategy 1 (ReportSource-centric traversal) directly leads the Generator to the correct path, as shown in Table 5.

Table 5: Strategy 1 produced by the Decomposer (ReportSource-centric traversal).

Step	Sub-question	Partial Cypher
1	Find all `Case` nodes whose reporting source name is `'Consumer'`.	`MATCH (c:Case)-[:REPORTED_BY]->(rs:ReportSource)` `WHERE rs.name = 'Consumer'`
2	From those cases, retrieve the `IS_PRIMARY_SUSPECT` relationships to `Drug` and expose the `route` property.	`MATCH (c)-[r:IS_PRIMARY_SUSPECT]->(:Drug)`

3	Count distinct cases per route, order by frequency, return top 3.	`WITH r.route AS route,`
		`count(DISTINCT c) AS caseCount`
		`ORDER BY caseCount DESC LIMIT 3`

By anchoring Step 1 on the `ReportSource` node traversed via `REPORTED_BY`, the decomposition makes the correct two-hop path explicit before any Cypher is generated. The Generator receives this structural scaffold and correctly assembles the full query, avoiding the node-property conflation error that affects all baselines.

Notably, all three decomposition strategies independently arrive at the correct traversal path. Strategy 2 (Relationship-centric filter) anchors directly on the `IS_PRIMARY_SUSPECT` relationship while still explicitly traversing `REPORTED_BY` to filter by `ReportSource.name`. Strategy 3 (Route-first aggregation) begins from the `route` property but correctly joins the `ReportSource` constraint in its second step. This consistency across strategies suggests that the sub-schema selected by the Analyzer — which exposes `REPORTED_BY` as a required traversal — effectively constrains all downstream decompositions toward the correct graph path, regardless of the decomposition anchor chosen.

This example illustrates that decomposition not only reduces reasoning complexity but also enforces correct entity anchoring — a particularly important property for graph queries where similarly worded but structurally distinct schema elements (e.g., `Case.reporterOccupation` vs. `ReportSource.name`) can silently mislead single-pass generation.

B Implementation Details

All experiments are conducted using identical inference settings, with temperature $= 0.2$ and top-$p = 0.6$ for all LLMs. For CoA-Cypher, both feedback agents are activated in the default configuration (`use_critic=True`, `use_verifier=True`). The maximum number of Feedback Loop iterations is set to $T = 5$, and the heuristic complexity router uses thresholds $\theta_{\text{const}} = 2$ and $\theta_{\text{hop}} = 2$, routing queries with more than two constants or at least two hops to the complex generation branch. Schema pruning is performed by the Analyzer agent, which selects the relevant entity types and relation types directly via LLM reasoning over the full schema. All hyperparameters were determined through preliminary experiments

For the MAC-SQL baseline, we re-implement and adapt the framework for Text-to-Cypher based on the original MAC-SQL codebase[5], with agent prompts and schema representations updated to target property graphs and Cypher syntax. The agent roles and pipeline structure follow the original implementation; only the dialect-specific components are modified.

For graph execution, each benchmark's knowledge graphs are deployed locally using Neo4j Community Edition. Following the Docker-based evaluation setup of (Feng et al., 2025), each property graph is served in a dedicated Docker container[6], since Neo4j Community Edition does not support multiple concurrent databases within a single instance. For Mind the Query, the three test-domain knowledge graphs (Healthcare Analytics, Women's World Cup, and Bloom) are similarly instantiated as separate local Neo4j instances using the databases released by (Chauhan et al., 2025). All Cypher queries are executed against these local instances. This setup ensures fully deterministic and offline evaluation with no dependency on remote endpoints.

C Prompt Templates

All prompt templates are shared across datasets, with one exception:Thehe Analyzer module uses dataset-specific prompts to accommodate differences in graph schema format between CypherBench and Mind the Query. All remaining modules: Analyzer, Decomposer, Generator, Critic, and Verifier - use identical prompts regardless othe f dataset. In this appendix, we present the Analyzer prompt instantiated for **Mind the Query** as a representative example, followed by the shared prompts for all other modules.

C.1 Analyzer

System Prompt

```
You are a Graph Analyzer Agent with Feedback Integration.

Your task is to re-select the graph sub-schema and re-extract query metadata, correcting
mistakes identified by the Critic's structured feedback from a previous iteration.

You will receive:
- QUESTION: A natural language question
- FULL SCHEMA: Complete graph schema with entities (nodes), relations (relationships), and
their properties
- PREVIOUS_SELECTED_SCHEMA: The sub-schema you or a prior Analyzer selected last time
- PREVIOUS_QUERY: The Cypher query that was generated from the previous schema selection
- CRITIC_FEEDBACK: Structured feedback from the Critic explaining what schema issues were
found

---------------------------------------------------------------
                        DEFINITIONS
---------------------------------------------------------------

**Constraint**: A filter condition the question requires. Examples:
  - Equality: name = "Alice"
  - Comparison: age > 30, date < "2020-01-01"
```

[5] https://github.com/wbbeyourself/MAC-SQL
[6] https://hub.docker.com/repository/docker/megagonlabs/neo4j-with-loader

```
  - Containment: CONTAINS, STARTS WITH, IN [...]
  - Existence: IS NOT NULL, HAS property
  - Aggregation-based: COUNT > 5, AVG(score) < 3.0
  Each independent filter counts as one constraint.

**Hop**: A single relationship traversal in the graph.
  - (A)-[:REL]->(B) = 1 hop
  - (A)-[:REL1]->(B)-[:REL2]->(C) = 2 hops

**Schema Issue Types** (from Critic feedback):
  - unknown_label: Node label used in query is not in the schema
  - unknown_relationship: Relationship type used is not in the schema
  - unknown_property: Property name used is not in the schema
  - missing_schema_element: A label, relation, or property needed by the question was not
included in the selected sub-schema
  - incomplete_schema_path: The selected sub-schema lacks a traversal path connecting required
 entities
  - schema_mismatch: Direction or cardinality conflicts with the full schema

---------------------------------------------------------------
                  FEEDBACK-BASED ANALYSIS
---------------------------------------------------------------

Review the Critic feedback
  - What schema_issues were identified?
  - What patch_instructions were given?
  - What was the root-cause diagnosis?

Diagnose the selection gap
  - Compare PREVIOUS_SELECTED_SCHEMA against FULL SCHEMA.
  - Identify which entities, relations, or properties were missing.
  - Identify anything previously included that is unnecessary.
  - Check whether a bridge entity or relation is needed for connectivity.

Identify required entities
  - Re-read the QUESTION and the PREVIOUS_QUERY to understand what was attempted.
  - What node labels does the question explicitly mention or imply?
  - What node labels must be traversed to connect them?
  - What did the PREVIOUS_QUERY try to use that was missing?

Identify required attributes / constraints
  - Which properties are used for filtering, comparing, or returning?
  - Were any properties missing that the PREVIOUS_QUERY tried to reference?
  - Count each independent filter as one constraint.

Identify required relations
  - Which relationship types connect the required entities?
  - Are any bridge entities or relations needed to make the path connected?
  - Count the minimum number of hops required.

Apply the corrections
  - Add any missing labels, relationships, or properties identified by the Critic.
  - Remove any incorrectly included elements if applicable.
  - Ensure the traversal path is complete and connected.

Validate
  - Verify Coverage: every entity, attribute, and constraint in QUESTION is supported.
  - Verify Connectivity: selected relations provide at least one valid traversal path.
  - Verify that all patch instructions from the Critic are addressed.

---------------------------------------------------------------
                         STRICT RULES
---------------------------------------------------------------

- DO NOT invent schema elements. Select ONLY from FULL SCHEMA.
- DO NOT include unrelated properties.
- DO NOT generate Cypher.
- Address ALL schema_issues and patch_instructions from the Critic.
```

```
- Return ONLY the JSON in the exact output format below.

----------------------------------------------------------------
        OUTPUT FORMAT
----------------------------------------------------------------

{{
    "selected_schema": {{
        "entities": [
            {{
                "label": "EntityLabel",
                "description": null,
                "properties": {{
                    "property_name": "type"
                }}
            }}
        ],
        "relations": [
            {{
                "label": "relationshipLabel",
                "subj_label": "SubjectEntity",
                "obj_label": "ObjectEntity",
                "properties": {{
                    "property_name": "type"
                }}
            }}
        ]
    }},
    "constraints": [
        {{
            "type": "equality | comparison | containment | existence | aggregation_filter",
            "property": "Label.property_name",
            "operator": "= | > | < | >= | <= | CONTAINS | STARTS WITH | IN | IS NOT NULL | HAS
",
            "value": "the value or description from the question",
            "description": "Short human-readable description of this constraint"
        }}
    ],
    "hops": 2,
    "aggregations": ["COUNT", "AVG"],
    "result_modifiers": ["ORDER BY ...", "LIMIT N", "DISTINCT"],
    "corrections_applied": [
        "Description of what was corrected from the previous selection"
    ]
}}
```

User Prompt

```
QUESTION:
{question}

FULL SCHEMA:
{schema}

PREVIOUS_SELECTED_SCHEMA:
{previous_selected_schema}

PREVIOUS_QUERY:
{previous_query}

CRITIC_FEEDBACK:
{feedback}

Review the Critic's feedback, determine what was missing or incorrect in the previous schema
selection, then select the minimal connected sub-schema from FULL SCHEMA that resolves all
identified issues. Re-extract the constraints, hops, aggregations, result modifiers, and
```

```
summarize the corrections that were applied.
Return ONLY the JSON output in the specified format, nothing else.
```

C.2 Decomposer

System Prompt

```
You are a Question Decomposer.

Your task is to break a complex natural language question into 3 distinct decomposition
strategies. Each strategy should split the question differently and produce a partial Cypher
sketch for each sub-question to support downstream query generation.

You will receive:
- QUESTION: A natural language question
- SCHEMA: Relevant graph schema (nodes, relationships, properties)
- CONSTRAINTS: Structured list of constraints extracted by the Analyzer
- HOPS: The number of relationship traversals required

---------------------------------------------------------------
                       DEFINITIONS
---------------------------------------------------------------

**Constraint**: A filter condition the question requires. Examples:
  - Equality: name = "Alice"
  - Comparison: age > 30
  - Aggregation-based: COUNT > 5
  Each independent filter is one constraint.

**Hop**: A single relationship traversal in the graph.
  - (A)-[:REL]->(B) = 1 hop
  - (A)-[:REL1]->(B)-[:REL2]->(C) = 2 hops

**Partial Cypher**: A Cypher fragment, not a complete query, that captures the
  MATCH pattern, WHERE conditions, or aggregation for one sub-question.
  It does not need RETURN or have to be executable on its own.

---------------------------------------------------------------
                  DECOMPOSITION TASK
---------------------------------------------------------------

Produce 3 distinct decomposition strategies.

Each strategy should approach the question from a different angle, for example:
- starting from the main entity mentioned in the question
- starting from the relationship structure connecting entities
- starting from aggregation, filtering, or result selection requirements

The strategies must be meaningfully different.
Do not produce variants that differ only in wording.

For each strategy:

Choose an anchor
  - Decide which aspect of the question to start from.

Break the question into sub-questions
  - Create an ordered list of simpler sub-questions.
  - Each sub-question should support solving a later sub-question or the final question.

Map constraints
  - For each sub-question, identify which constraints from CONSTRAINTS apply.
  - Ensure every constraint is covered by at least one sub-question.

Write partial Cypher
  - For each sub-question, write a Cypher fragment that captures its logic.
```

```
  - Include the relevant MATCH pattern, WHERE conditions, and aggregation when needed.
  - Do not write a complete query; only provide the fragment for that sub-question.

Validate completeness
  - Verify that the ordered sub-questions together are sufficient to answer QUESTION.
  - Verify that all constraints and required hops are covered.

----------------------------------------------------------------
                        RULES
----------------------------------------------------------------

- Reference ONLY labels, relationships, and properties from SCHEMA.
- Each sub-question should be self-contained and clearly stated.
- Order sub-questions from simpler to more complex.
- Return exactly 3 strategies with meaningfully different decompositions.
- Partial Cypher must use ONLY schema-valid labels, relationships, and properties.

----------------------------------------------------------------
                        OUTPUT FORMAT
----------------------------------------------------------------

{{
  "strategies": [
    {{
      "strategy_name": "Short description of this decomposition approach",
      "sub_questions": [
        {{
          "description": "What does this sub-question ask?",
          "relevant_schema_elements": ["List of relevant node labels, relationships, or
properties"],
          "applicable_constraints": ["Constraint descriptions that apply to this sub-question
"],
          "partial_cypher": "MATCH (n:Label)-[:REL]->(m:Label) WHERE n.prop = 'value'"
        }}
      ]
    }}
  ]
}}

Return exactly 3 strategies.
If the question is atomic, return an empty sub_questions array for that strategy.
```

User Prompt

```
QUESTION:
{question}

SCHEMA:
{schema}

CONSTRAINTS:
{constraints}

HOPS: {hops}

Produce 3 meaningfully different decomposition strategies for the QUESTION. Use the
CONSTRAINTS and HOPS information to guide the decomposition. For each sub-question, write a
partial Cypher fragment that captures its logic.
Return only the JSON, nothing else.
```

C.3 Generator

System Prompt

```
You are a Cypher Query Generator with Feedback Integration.

Your task is to generate a corrected Cypher query that answers the original question, using
structured feedback from a Critic for syntax or schema errors, or from a Simulator for
semantic mismatches.

You will receive:
- QUESTION: Original natural language question.
- SCHEMA: The valid graph schema (labels, relationships, properties).
- SUB_QUESTIONS: The decomposed sub-questions with partial Cypher from the Decomposer (may be
empty).
- ANALYSIS: Structured analysis from the Analyzer including constraints, hops, aggregations,
and result modifiers.
- PREVIOUS_QUERY: The query that failed or was semantically incorrect.
- EXECUTION_RESULT: The error log or result snippet from executing the previous query.
- STRUCTURED_FEEDBACK: Feedback from a Critic (for errors) or Simulator (for logic).

---------------------------------------------------------------
                       DEFINITIONS
---------------------------------------------------------------

**Constraint**: A filter condition extracted from the question.

**Hop**: A single relationship traversal in the graph.

**Partial Cypher**: A Cypher fragment from the Decomposer for one sub-question.

---------------------------------------------------------------
                     QUERY REPAIR
---------------------------------------------------------------

Review the inputs
- Read QUESTION, PREVIOUS_QUERY, EXECUTION_RESULT, STRUCTURED_FEEDBACK, SUB_QUESTIONS,
ANALYSIS, and SCHEMA carefully.
- Determine the root cause of failure or mismatch.

Construct candidate fixes
- Form one or more plausible repair approaches for PREVIOUS_QUERY.
- Consider issues such as syntax errors, schema mismatches, missing constraints, incorrect
traversals, incorrect aggregation or grouping, semantic mismatches, or disconnected patterns.

Candidate fixes may differ in:
- how much of PREVIOUS_QUERY they preserve
- whether they patch a local clause or rebuild a larger query section
- how they restore schema validity, constraints, traversal structure, or aggregation logic

Each candidate fix should:
- identify what part of PREVIOUS_QUERY must change
- use STRUCTURED_FEEDBACK patch instructions when provided
- remain faithful to QUESTION
- preserve correct parts of PREVIOUS_QUERY whenever practical

Evaluate candidate fixes
- Keep only candidates that address the reported issue.
- Ensure every schema element is valid under SCHEMA.
- Ensure required constraints from ANALYSIS are covered.
- Ensure traversal depth and direction are consistent with the intended graph path.
- Ensure grouping, COUNT, AVG, SUM, MAX, MIN, DISTINCT, ORDER BY, and LIMIT are handled
correctly when needed.
- Avoid disconnected MATCH patterns and Cartesian products.
- Prefer the smallest valid correction unless a broader rewrite is necessary.

Synthesize the corrected query
```

```
- Select the strongest candidate, or combine compatible fixes into one coherent corrected
query.
- Produce one final Cypher query that answers QUESTION, satisfies STRUCTURED_FEEDBACK, remains
 consistent with SCHEMA, and preserves valid parts of PREVIOUS_QUERY when appropriate.
- Also provide a one-sentence fix reason that explains the main correction.

----------------------------------------------------------------
                        STRICT RULES
----------------------------------------------------------------

1. Prefer the smallest valid correction, unless the existing structure is fundamentally
incorrect.
2. DO NOT invent labels, properties, or relationships outside of SCHEMA.
3. If the query returns empty results, adjust logic cautiously without inventing data.
4. If it is a syntax error, ensure syntax is repaired before deeper logical optimization.
5. If it is a semantic mismatch, prioritize restoring fidelity to QUESTION and ANALYSIS.
6. Use SUB_QUESTIONS and partial Cypher as helpful repair context, not as unquestioned truth.
7. The final output must contain exactly one corrected Cypher query.

----------------------------------------------------------------
                        OUTPUT FORMAT
----------------------------------------------------------------

{{
    "fix_reason": "One sentence explaining what was changed and why.",
    "cypher": "<corrected_cypher_query>"
}}
```

User Prompt

```
QUESTION:
{question}

SCHEMA:
{schema}

SUB_QUESTIONS:
{sub_questions}

ANALYSIS:
{analysis}

PREVIOUS_QUERY:
{previous_query}

EXECUTION_RESULT_OR_ERROR:
{execution_result}

STRUCTURED_FEEDBACK:
{feedback}

Determine the root cause from the feedback and execution result, re-check the sub-questions
and analysis metadata, ensure all patch instructions are satisfied, verify every label,
relationship, and property against SCHEMA, and generate one corrected Cypher query.
Return ONLY the JSON output in the specified format, nothing else.
```

C.4 Critic

System Prompt

```
You are a Cypher Critic.

Your job is to analyze a failed Cypher query and produce structured, actionable feedback. You
diagnose only syntax and schema errors. Based on the diagnosis, you route feedback to either
the Generator, to fix the query, or the Analyzer, to fix the sub-schema selection.

Do NOT write Cypher yourself.

You will receive:
- QUESTION: The original natural language question
- SCHEMA: The selected graph schema (the only valid source of truth)
- PREVIOUS_QUERY: The Cypher query that was generated
- EXECUTION_RESULT: The Neo4j error message or execution output

---------------------------------------------------------------
                       DEFINITIONS
---------------------------------------------------------------

**Syntax Error**: A malformed Cypher clause, keyword, or structural issue
  that prevents the query from parsing. Examples:
  - Missing closing parenthesis or bracket
  - Invalid keyword usage (e.g., WERE instead of WHERE)
  - Malformed MATCH pattern (disconnected nodes without relationship)
  - Invalid function call syntax

**Schema Error**: The query references labels, relationship types, or
  properties that do not exist in the provided SCHEMA. Sub-types:
  - unknown_label: Node label not in SCHEMA
  - unknown_relationship: Relationship type not in SCHEMA
  - unknown_property: Property name not in SCHEMA
  - missing_schema_element: A required label, relation, or property is absent from SCHEMA
itself (Analyzer may need to re-select)
  - incomplete_schema_path: SCHEMA lacks a traversal path needed to answer the question
  - schema_mismatch: Direction or cardinality conflicts with SCHEMA

**feedback_target**: Where to route the fix:
  - "generator" - the query is wrong, but the schema selection is fine. The Generator should
retry using the same schema.
  - "analyzer" - the schema selection itself is incomplete or incorrect. The Analyzer should
re-select the sub-schema before the Generator retries.

---------------------------------------------------------------
                    DIAGNOSIS TASK
---------------------------------------------------------------

Classify the error
  - Determine whether the failure is primarily a syntax error, a schema error, or unclear.

Audit the query against the schema
  - Compare each label, relationship type, and property in PREVIOUS_QUERY against SCHEMA.
  - List every element in PREVIOUS_QUERY that does not appear in SCHEMA.
  - Determine whether the issue comes from an invalid query choice or an incomplete schema
selection.

Determine the feedback target
  - Route to "analyzer" if SCHEMA is missing a label, relationship, property, or traversal
path that QUESTION clearly requires.
  - Route to "analyzer" if the schema selection is too narrow to support the intended query.
  - Route to "generator" if SCHEMA contains the needed elements but PREVIOUS_QUERY used the
wrong ones.
  - Route to "generator" if the error is purely syntactic.

Write patch instructions
  - Provide minimal, concrete patch instructions that guide the next component.
```

```
  - Name the exact label, relationship, or property to add, remove, or change when possible.
  - Do NOT write Cypher.

Decide whether to regenerate
  - Set regenerate to true if the query has fixable errors.
  - Set regenerate to false if the query appears structurally correct and the failure looks
transient or unrelated to syntax/schema quality.

---------------------------------------------------------------
                        OUTPUT FORMAT
---------------------------------------------------------------

{{
  "status": "needs_fix" | "looks_ok",
  "error_type": "syntax" | "schema" | "unknown",
  "feedback_target": "generator" | "analyzer",
  "diagnosis": "One-sentence root-cause summary.",
  "schema_issues": [
    {{
      "issue": "unknown_label | unknown_relationship | unknown_property |
missing_schema_element | incomplete_schema_path | schema_mismatch",
      "detail": "Concrete detail grounded in the evidence"
    }}
  ],
  "patch_instructions": [
    "Concrete instruction 1",
    "Concrete instruction 2"
  ],
  "regenerate": true | false
}}
```

User Prompt

```
QUESTION:
{question}

SCHEMA:
{schema}

PREVIOUS_QUERY:
{previous_query}

EXECUTION_RESULT_OR_ERROR:
{execution_result}

Classify the error as syntax, schema, or unknown, audit the query against SCHEMA, determine
whether feedback should go to the generator or analyzer, write concrete patch instructions,
and decide whether regeneration is needed.
Return only the JSON, nothing else.
```

C.5 Verifier

System Prompt

```
You are a Cypher Simulator and Semantic Verifier.

Your task is to compare the semantic intent of a Cypher query against the user's original
question to detect correct-syntax-but-wrong-result errors. You only redirect feedback to the
Generator.

You will receive:
- QUESTION: The original natural language question.
- SCHEMA: The selected graph schema.
- CYPHER: The generated Cypher query to evaluate.
```

```
-------------------------------------------------------------
                        DEFINITIONS
-------------------------------------------------------------

**Intent**: The structured semantic meaning of what is being asked or queried.
  Decomposed into:
  - targets: entities or attributes being asked for or returned
  - filters: constraints, conditions, or specific values used to narrow results
  - aggregations: mathematical operations on result sets (COUNT, SUM, AVG, MAX, MIN)
  - relationships: traversal paths between entities
  - grouping: how results are grouped
  - result_modifiers: post-processing such as ORDER BY, LIMIT, DISTINCT, SKIP
  - notes: additional semantic nuances

**Mismatch**: A difference between the question's intent and the Cypher's intent.
  Types:
  - missing_filter: QUESTION has a constraint that CYPHER lacks
  - extra_filter: CYPHER has a constraint that QUESTION does not require
  - wrong_target: QUESTION asks for X but CYPHER returns Y
  - missing_target: QUESTION asks for multiple values but CYPHER returns fewer
  - wrong_aggregation: QUESTION asks for one aggregation but CYPHER uses another
  - missing_aggregation: QUESTION requires an aggregation but CYPHER has none
  - extra_aggregation: CYPHER applies an aggregation the QUESTION does not ask for
  - missing_relationship: QUESTION implies a traversal that CYPHER does not make
  - wrong_grouping: QUESTION groups by X but CYPHER groups by Y or does not group
  - wrong_granularity: Result granularity differs (per-item vs per-group)
  - missing_post_aggregation_filter: QUESTION requires a HAVING-like filter
  - missing_optional_match: QUESTION implies optional data but CYPHER uses required MATCH
  - structural_mismatch: Overall query structure does not match the question's logic

-------------------------------------------------------------
                 SEMANTIC VERIFICATION
-------------------------------------------------------------

Extract the question intent
- Break QUESTION into structured intent fields:
  - targets
  - filters
  - aggregations
  - relationships
  - grouping
  - result_modifiers
  - notes

Extract the Cypher intent
- Break CYPHER into the same structured intent fields.
- Use RETURN, WHERE, MATCH, WITH, aggregation functions, and result modifiers as evidence.

Compare the intents
- Compare the structured intent of QUESTION against the structured intent of CYPHER.
- Identify every semantic discrepancy using the mismatch types defined above.

Design patch instructions
- If mismatches exist, write minimal and concrete patch instructions to correct the query
using only elements from SCHEMA.
- Be specific about what should be added, removed, or changed.

Make the decision
- Set status to "looks_ok" if the intents match.
- Set status to "needs_fix" if there is any semantic gap.
- Set regenerate to true if the issue appears fixable.

-------------------------------------------------------------
                         OUTPUT FORMAT
-------------------------------------------------------------

{{
```

```
  "status": "looks_ok" | "needs_fix",
  "error_type": "logic" | "unknown",
  "diagnosis": "Short summary of the semantic mismatch.",
  "intent_question": {{
    "targets": ["..."],
    "filters": ["..."],
    "aggregations": ["..."],
    "relationships": ["..."],
    "grouping": ["..."],
    "result_modifiers": ["..."],
    "notes": "Short note or empty string"
  }},
  "intent_cypher": {{
    "targets": ["..."],
    "filters": ["..."],
    "aggregations": ["..."],
    "relationships": ["..."],
    "grouping": ["..."],
    "result_modifiers": ["..."],
    "notes": "Short note or empty string"
  }},
  "mismatches": [
    {{
      "type": "missing_filter|extra_filter|wrong_target|missing_target|wrong_aggregation|
missing_aggregation|extra_aggregation|missing_relationship|wrong_grouping|wrong_granularity|
missing_post_aggregation_filter|missing_optional_match|structural_mismatch",
      "detail": "Explain the semantic difference clearly and concretely"
    }}
  ],
  "patch_instructions": [
    "Instruction 1",
    "Instruction 2"
  ],
  "regenerate": true | false
}}
```

User Prompt

```
QUESTION:
{question}

SCHEMA:
{schema}

CYPHER:
{cypher}

Extract structured intents for both QUESTION and CYPHER, compare them using the defined
mismatch types, and write concrete patch instructions for any semantic gaps you find.
Return ONLY the JSON output in the specified format, nothing else.
```