Adaptive Text2GQL: Integrating Structural Twig Linking and Evolutionary In-Context Learning

Fang Niu, Chaokun Wang*, **Hang Zhang, Songyao Wang**
Tsinghua University, China
{nf21, zhanghang24, wangsong23}@mails.tsinghua.edu.cn
chaokun@tsinghua.edu.cn

Abstract

While large language models have revolutionized Text-to-SQL tasks, translating natural language into Graph Query Languages (Text2GQL) remains underexplored due to the topological heterogeneity and syntactic diversity of graph query languages (e.g., Cypher, Gremlin, SPARQL). Existing approaches often struggle with *structural hallucinations* and lack adaptability in cold-start scenarios. In this paper, we present a unified, training-free Text2GQL framework. First, **Structural Twig Linking** elevates schema grounding to the identification of semantic substructures ("twigs"), providing robust topological priors. Second, addressing data scarcity, **Evolutionary In-Context Learning** operates in a *Tabula Rasa* setting to implicitly construct a self-growing repository of verified examples driven by syntactic utility. Finally, our **Adversarial Execution-Guided Correction** agent enforces fidelity through synergistic static critique and dynamic verification. Experiments demonstrate significant improvements over baselines in both accuracy and executability across diverse GQLs. The code is available at https://github.com/nf202/Text2Graph.

1 Introduction

Graph-structured data has become ubiquitous in diverse applications such as social networks (Singh et al., 2024), knowledge graphs (Hogan et al., 2021), recommender systems (Zheng et al., 2025a,b), GraphRAG (Zheng et al., 2026), and multimodal search (Kennedy et al., 2005). However, substantial high-value data remains "siloed" across disparate database engines (e.g., Neo4j, JanusGraph, RDF stores), each governed by distinct and often incompatible query standards. The ecosystem is fragmented into declarative languages like **Cypher** (Francis et al., 2018) and **SPARQL** (Pérez et al., 2009), procedural traversals like **Gremlin** (Rodriguez, 2015), and the emerging **ISO/IEC 39075:2024** standard (ISO/IEC JTC 1/SC 32, 2024). This heterogeneity creates a "Tower of Babel" scenario: a single analytical intent (e.g., "find co-authors") requires radically different syntactic expressions depending on the underlying backend. Consequently, there is an urgent demand for a unified interface capable of "cross-language generalization"—faithfully translating a single natural language question into multiple target graph query languages (GQLs) to unlock data across heterogeneous graph systems.

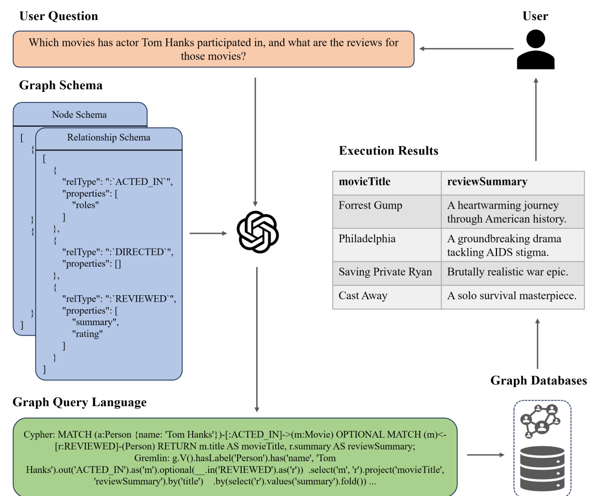

Figure 1: An illustrative example of the **Text2GQL** workflow. The framework translates a natural language question into heterogeneous graph queries guided by the schema, accurately retrieving structured results from the databases.

Text-to-Graph Query Language (Text2GQL) task aims to bridge this gap by automating the translation of natural language questions into GQLs that accurately reflect user intent, as illustrated in Fig. 1. While early rule-based approaches lacked scalability, the advent of Large Language Models (LLMs) has revolutionized this domain, superseding rigid

* Corresponding author.

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 17663–17680
July 2-7, 2026 ©2026 Association for Computational Linguistics

templates with context-aware generation capabilities via Chain-of-Thought (CoT) reasoning (Wei et al., 2022). However, deploying general-purpose LLMs as a unified interface for fragmented graph databases remains non-trivial due to the strict topological constraints and the rigid, heterogeneous grammars of diverse GQLs.

We identify four primary challenges in robust, cross-platform Text2GQL generation: **(C1) Schema Grounding Complexity:** Unlike tabular schemas, graph schemas involve dense, directed dependencies. Accurately linking NL (natural language) mentions to specific nodes or edges requires navigating complex topological contexts unique to each database instance. **(C2) Paradigm Incompatibility:** The distinct paradigms of target languages—declarative pattern matching (Cypher/SPARQL) vs. procedural traversal (Gremlin)—hinder the development of a unified generation model that generalizes across backends. **(C3) Cold-Start Adaptation:** The scarcity of high-quality, parallel cross-language benchmarks impedes supervised fine-tuning, necessitating effective few-shot strategies for unseen domains and languages. **(C4) Structural Hallucination:** Due to sparse GQL training data, LLMs often correctly infer user intent but fail to ground it in valid syntax, yielding queries that are *logically consistent yet topologically or syntactically invalid.*

To address these challenges, we propose a comprehensive, training-free Text2GQL framework structured around three synergistic modules. First, to jointly resolve **C1** and **C2**, we propose **Structural Twig Linking** (Sec. 4.1). This mechanism unifies semantic grounding with language-specific topological instantiation, mapping NL tokens directly to valid query skeletons ("twigs") that respect both the dense connectivity of graph schemas and the specific syntax of the target GQL. Second, addressing **C3**, we present **Evolutionary In-Context Learning** (Sec. 4.2), a self-adaptive paradigm that implicitly constructs a domain-specific shot repository from a *Tabula Rasa* state, enabling robust cold-start adaptation without relying on pre-annotated datasets. Finally, to mitigate **C4**, we implement **Adversarial Execution-Guided Correction** (Sec. 4.3), an autonomous agent that synergizes static adversarial critique with dynamic runtime verification to rigorously filter structural hallucinations. Our main contributions are summarized as follows:

- **Formalization of Cross-Language Text2GQL:** We present a formal definition of the Text2GQL task (Sec. 2), establishing a unified formulation that incorporates heterogeneous schema constraints across diverse query languages.
- **A Unified Algorithmic Framework:** We propose a modular, training-free framework (Sec. 3) that synergizes structural twig linking, evolutionary in-context learning, and adversarial execution-guided correction (Sec. 4). This framework effectively bridges the gap between semantic intent and topological correctness across different GQLs.
- **Empirical Validation:** We conduct extensive experiments on multiple graph query languages (Sec. 5). The results demonstrate that our framework significantly enhances generation accuracy and robustness compared to existing baselines, validating its efficacy for natural language interaction with heterogeneous graph databases.

2 Preliminaries

In this section, we formalize the **Text2GQL** definition and define two core structures in our framework: the **Structural Twig** (a unit of topological grounding) and the **Evolutionary Shot Repository** (a dynamic repository for cold-start adaptation).

2.1 Text2GQL Formulation

We define **Text2GQL** as a conditional generation problem translating a natural language question Q into a set of GQLs Y (e.g., Cypher, Gremlin). Let $\mathcal{S} = (N, R, P, C)$ denote the graph schema, comprising node types N, relationship types R, properties P, and constraints C. Given a Large Language Model parameterized by θ, the objective is to synthesize the query $\hat{Y}$ by conditioning the model on the input question and schema context:

$$\hat{Y} = \text{LLM}_\theta(Q, \mathcal{S}) \quad (1)$$

where the generation is constrained by the topological structure in $\mathcal{S}$ to ensure syntactic validity and semantic fidelity.

2.2 Structural Twig

To bridge the granularity gap between linear text tokens and graph structures, we propose the concept of a **Structural Twig** (denoted as h). Conceptually, a Twig represents a minimal, semantically complete subgraph unit derived from the schema $\mathcal{S}$. Unlike isolated vocabulary terms, a Twig captures

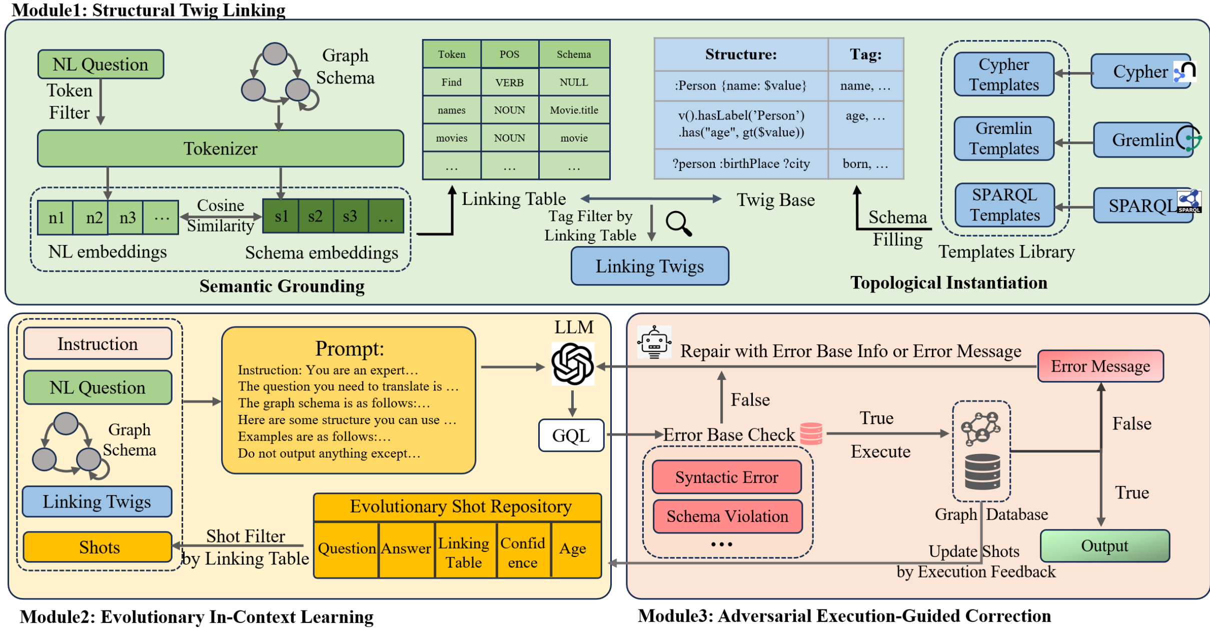

Figure 2: Overview of the **Adaptive Text2GQL** framework. The system comprises three core modules: (1) **Structural Twig Linking**, which constructs valid query skeletons by synergizing semantic grounding and topological instantiation; (2) **Evolutionary In-Context Learning**, which dynamically evolves a shot repository based on syntactic utility; and (3) **Adversarial Execution-Guided Correction**, an agent-based loop that refines queries via static critique and dynamic verification.

the atomic topological skeleton of the graph. For instance, it may manifest as an **Atomic Triplet** (n_h, r, n_t) describing a directed relationship r between head n_h and tail n_t, or a **Property Binding** (n, p) that explicitly associates a node type n with an attribute p. By encapsulating local connectivity rules, a collection of Twigs $H = \{h_1, h_2, ...\}$ serves as the foundational building blocks, allowing the model to assemble complex query structures from valid topological components.

2.3 Evolutionary Shot Repository

To address *Tabula Rasa* (cold-start) scenarios, we propose the concept of an **Evolutionary Shot Repository** (denoted as $\mathcal{L}$). This repository functions as a dynamic repository that evolves during the inference process, distinguishing itself from static pre-annotated datasets. Starting from an empty state, the repository progressively accumulates **verified Q-A pairs**—instances where the generated query has been confirmed executable by the database engine. By preserving these concrete, history-proven examples, the system can retrieve relevant shots from this growing pool based on similarity. These examples serve as validated structural precedents, guiding the model's adherence to complex GQL syntax without external supervision.

3 Overview

We present **Adaptive Text2GQL**, a unified framework integrating three synergistic modules: **Structural Twig Linking**, **Evolutionary In-Context Learning**, and **Adversarial Execution-Guided Correction**. As shown in Fig. 2, this closed-loop architecture transcends linear pipelines by combining structural priors, dynamic context, and verification feedback to optimize generation fidelity.

Structural Twig Linking. To bridge the gap between text and graph, this module synthesizes *Twigs*—semantic subgraphs representing local connectivity. By mapping NL tokens to valid query skeletons via semantic and topological grounding, it provides robust structural priors that effectively constrain the LLM's search space.

Evolutionary In-Context Learning. Targeting cold-start scenarios, this module employs a self-adaptive *Tabula Rasa* paradigm. Unlike static retrieval, it builds an evolving shot repository based on *syntactic utility*, progressively accumulating execution-verified patterns to guide generation without pre-annotated data.

Adversarial Execution-Guided Correction. Functioning as an autonomous agent, this module ensures reliability via a dual-phase loop: a *Static Critic* for preemptive filtering and a *Dynamic Veri-*

fier leveraging runtime feedback. This mechanism repairs defective queries while reinforcing the evolutionary shot repository with validity signals.

4 Methodology

This section details the internal mechanics of our **Adaptive Text2GQL** framework. We decompose the unified generation pipeline into three synergistic paradigms: **Structural Twig Linking** (Sec. 4.1), **Evolutionary In-Context Learning** (Sec. 4.2), and **Adversarial Execution-Guided Correction** (Sec. 4.3). Each component is elaborated in the following subsections.

4.1 Structural Twig Linking

This module operationalizes the concept of Structural Twig defined in Sec. 2.2. The objective is to map the question Q with a set of validated schema substructures, denoted as *Twigs*. This process is executed through a two-stage pipeline: *Semantic Grounding*, which identifies atomic schema elements, and *Topological Instantiation*, which assembles these atoms into valid GQL patterns.

4.1.1 Semantic Grounding

The primary objective of this stage is to construct an **Atomic Mapping Table** $\mathcal{M}$, which serves as the foundational anchor between natural language mentions and atomic graph elements (nodes, edges, properties). We formulate this grounding process as a mapping function $\Phi : Q \times \mathcal{S} \to \mathcal{M}$.

Linguistic Parsing and Filtering. To identify candidate keys for $\mathcal{M}$, we first apply a linguistic pipeline to extract content-bearing tokens from the raw query Q. Let $\mathbf{T}_{\text{raw}} = \text{Tokenize}(Q)$. We assign Part-of-Speech (POS) tags to generate $\mathbf{T}_{\text{tagged}} = \{(t_i, l_i)\}$, where t_i represents the token and l_i denotes the assigned POS tag. To reduce noise, we define a filtering operator to retain only semantically significant tokens while discarding functional stopwords:

$$\begin{aligned}\mathbf{T}_{\text{filtered}} = \{(t_i, l_i) \in \mathbf{T}_{\text{tagged}} \mid l_i \notin \{\text{DET, ADP, PUNCT}\} \\ \wedge\, t_i \notin \mathbb{V}_{\text{stop}}\}\end{aligned} \tag{2}$$

This step imposes a linguistic prior, ensuring that the subsequent population of $\mathcal{M}$ focuses strictly on potential entity and relationship mentions.

Embedding-Based Alignment. To determine the values of $\mathcal{M}$, we bridge the lexical gap by projecting both tokens and schema elements $\mathcal{S}$ into a shared d-dimensional semantic space. Let $\mathbf{v}_t$ and $\mathbf{v}_s$ denote the vector representations. We compute the pairwise alignment score using the cosine similarity:

$$\sigma(t, s) = \frac{\mathbf{v}_t \cdot \mathbf{v}_s}{\|\mathbf{v}_t\| \, \|\mathbf{v}_s\|} \tag{3}$$

Finally, the linkage for each token $t_i \in \mathbf{T}_{\text{filtered}}$ in the table is resolved via thresholding:

$$\mathcal{M}[t_i] = \begin{cases} \arg\max_{s \in \mathcal{S}} \sigma(t_i, s), & \text{if } \max_s \sigma(t_i, s) \geq \delta \\ \text{NULL}, & \text{otherwise} \end{cases} \tag{4}$$

where δ is a confidence threshold.

4.1.2 Topological Instantiation

While the former step identifies isolated vertices, this part assembles them into coherent *Twigs*—structural skeletons that reflect the native topology of the target GQL.

Template Library Construction. We define a library of language-specific structural templates, denoted as $\mathcal{P}_{\text{GQL}}$. Each template $p \in \mathcal{P}_{\text{GQL}}$ represents a high-frequency query pattern (e.g., traversal chains or aggregations) containing abstract slots $\square$ for schema injection. Formally:

$$\mathcal{P}_{\text{Cypher}} \ni p_{\text{triplet}} = \underbrace{(:\square)}_{\text{source}} - \underbrace{[\square]}_{\text{edge}} \to \underbrace{(:\square)}_{\text{target}} \tag{5}$$

Similar templates are defined for Gremlin (e.g., `g.V().has().out()`) and SPARQL (e.g., `?s ?p ?o`). We present a comprehensive catalog of these structural templates, covering diverse patterns such as attribute filtering, multi-hop traversals, and aggregations, in **Appendix B**.

Twig Generation via Instantiation. We generate candidate twigs by instantiating templates with the grounded elements from $\mathcal{S}$. The instantiation function $\text{Fill}(p, \mathcal{S})$ injects valid schema combinations into the slots of p:

$$\begin{aligned}\mathcal{H}_{\text{full}} = \bigcup_{p \in \mathcal{P}} \Big\{ p[s_1, \ldots, s_k] \mid s_i \in \mathcal{S} \\ \wedge \text{Valid}(s_1, \ldots, s_k) \Big\}\end{aligned} \tag{6}$$

where $\text{Valid}(\cdot)$ checks consistency against the graph schema constraints (e.g., domain and range of edges).

Relevance Scoring and Selection. To retrieve the most contextually relevant twigs, we employ a dual-tag scoring mechanism. Each candidate twig $h \in \mathcal{H}_{\text{full}}$ is characterized by: 1) **Schema Tags** $T_s(h)$: The set of schema elements contained in

the twig. 2) **Lexical Tags** $T_l(h)$: The set of NL keywords statistically associated with the pattern.

We compute a hybrid relevance score $\text{Score}(h)$ that balances schema alignment confidence (via $\mathcal{M}$) and lexical context matching:

$$\text{Score}(h) = \gamma \underbrace{\frac{|T_s(h) \cap \text{Values}(\mathcal{M})|}{|T_s(h)|}}_{\text{schema consistency}} + (1-\gamma) \underbrace{\sum_{w \in T_l(h)} \text{IDF}(w) \cdot \mathbb{I}(w \in \mathbf{T}_{\text{filtered}})}_{\text{lexical resonance}} \tag{7}$$

Here, γ is a weighting hyperparameter, and $\text{IDF}(w)$ penalizes common terms. Finally, we select the top-K twigs. The algorithm is shown in Appendix A.1.

4.2 Evolutionary In-Context Learning

We propose **Evolutionary In-Context Learning**, a module designed to construct and refine a repository of few-shot examples dynamically during the inference lifecycle. Operating in a cold-start setting (initially $\mathcal{L}_0 = \emptyset$), the system progressively builds a repository of valid query patterns based on execution feedback. This process transforms the shot repository from a static database into a self-evolving repository driven by syntactic utility.

The process iterates through four phases: repository structure maintenance, utility-driven selection, structure-augmented generation, and feedback-based evolution.

4.2.1 Evolutionary Shot Repository Structure

We define the shot repository $\mathcal{L}$ as a dynamic set of historical interaction records. Each entry $e_i \in \mathcal{L}$ is formalized as a tuple:

$$e_i = (\mathcal{C}_i, \rho_i, \tau_i)$$

- **Structural Content** $\mathcal{C}_i = (Q_i, G_i, \mathcal{M}_i)$: Contains the NL question Q_i, the generated GQL G_i, and the schema mapping table $\mathcal{M}_i$.
- **Syntactic Utility** $\rho_i \in [0, 1]$: A scalar metric representing the reliability of the shot, initialized upon entry and modulated by execution outcomes.
- **Evolutionary Age** $\tau_i \in \mathbb{N}$: A counter tracking the lifespan of the shot to facilitate temporal decay of outdated patterns.

4.2.2 Utility-Driven Shot Selection

For an incoming question Q_{new} with schema grounding $\mathcal{M}_{\text{new}}$, the system retrieves a subset of examples $\mathcal{E}_{\text{shots}} \subset \mathcal{L}$ that maximize both topological relevance and structural validity. The relevance score for each candidate e_i is computed as:

$$\text{Score}(e_i) = \alpha \cdot \text{SchemaSim}(\mathcal{M}_{\text{new}}, \mathcal{M}_i) + \beta \cdot \rho_i \cdot e^{-\lambda \tau_i} \tag{8}$$

This scoring function aggregates two critical dimensions:

- **Topological Similarity (SchemaSim):** Measures the overlap of referenced schema elements (nodes and edges), prioritizing examples that operate on similar graph sub-structures.
- **Time-Decayed Utility ($\rho_i \cdot e^{-\lambda \tau_i}$):** Prioritizes examples with high historical success rates (ρ_i) while penalizing older entries via the decay factor λ, ensuring the model adapts to recent feedback.

The selection process retrieves the top-K candidates based on this score, as detailed in Appendix A.2.

4.2.3 Structure-Augmented Generation

The selected shots $\mathcal{E}_{\text{shots}}$ serve as in-context demonstrations for LLMs. The generation model is conditioned on a composite prompt:

$$G_{\text{gen}} = \text{LLM}_\theta(I, \mathcal{S}, \mathcal{H}, \mathcal{E}_{\text{shots}}, Q_{\text{new}}) \tag{9}$$

where I denotes the task-specific system instruction guiding the generation behavior, $\mathcal{S}$ represents the schema definition, and $\mathcal{H}$ signifies the structural twigs (as defined in Sec. 4.1). By observing $\mathcal{E}_{\text{shots}}$, the model infers the valid syntactic structures and logical compositions required for the target GQL.

4.2.4 Feedback-Based Evolution

The core of the evolutionary mechanism is the continuous update of the repository based on execution feedback (derived from the Adversarial Execution-Guided Correction module in Sec. 4.3).

Utility Update. The syntactic utility ρ_i of the shots used in a generation step is updated based on whether the generated query G_{gen} successfully executes:

$$\rho_i^{(t+1)} = \begin{cases} \rho_i^{(t)} + \eta(1 - \rho_i^{(t)}) & \text{if } G_{\text{gen}} \text{ is valid} \\ \rho_i^{(t)} - \eta \rho_i^{(t)} & \text{if } G_{\text{gen}} \text{ fails} \end{cases} \tag{10}$$

where η is the learning rate. This reinforcement mechanism amplifies the influence of shots that

contribute to valid structural generation. Simultaneously, the age of all shots increments: $\tau^{(t+1)} = \tau^{(t)} + 1$.

Repository Construction. Following the GQL generation, the repository evolves through induction and eviction:

- **Induction:** If G_{gen} is verified as valid, the pair $(Q_{\text{new}}, G_{\text{gen}})$ is instantiated as a new shot with initial utility ρ_{init} and age $\tau = 0$, and then added to $\mathcal{L}$.
- **Eviction:** To maintain the repository size within capacity $|\mathcal{L}|_{\max}$, we remove the entry with the lowest current weighted score ($\rho_i \cdot e^{-\lambda\tau_i}$).

This cycle ensures that the repository implicitly constructs a domain-specific dataset of high-utility structural priors effectively adapting to the target database without manual supervision.

4.3 Adversarial Execution-Guided Correction

To guarantee the syntactic validity and semantic fidelity of the generated queries, we propose an **Adversarial Execution-Guided Correction (AEGC)** module. Functioning as an autonomous *Correction Agent*, this module orchestrates a closed-loop refinement process that synergizes static structural analysis with dynamic runtime verification. The correction lifecycle proceeds through the following two synergistic phases.

4.3.1 Adversarial Static Critique

The process initiates with a **Static Critic**, designed to preemptively filter structural errors prior to interaction with the database engine. This mechanism operates via two key steps:

- **Error Base Verification.** We utilize a curated error base $\mathcal{B}_{\text{err}}$, which encapsulates a hybrid set of *Regular Expression (Regex) validators* for syntax checking and *Database-specific anti-patterns* (e.g., missing colons in Cypher labels or invalid traversal steps in Gremlin). A detailed catalog of the error base patterns is provided in Appendix C.)
- **Preemptive Refinement.** The agent statically evaluates the generated query G_{gen} against $\mathcal{B}_{\text{err}}$. The detection of a matching error signature triggers an immediate, internal refinement, allowing the agent to rectify explicit violations without incurring the latency of database execution.

4.3.2 Dynamic Execution Verification

Following static filtration, the **Dynamic Verifier** assesses the semantic correctness of the refined query within the actual database environment.

- **Execution & Feedback Capture.** The query is executed against the schema $\mathcal{S}$. We classify execution outcomes into three distinct contexts: *Syntactic Errors* (parser violations), *Semantic Errors* (valid syntax yielding empty or incorrect results), and *Runtime Errors* (resource constraints).
- **Iterative Repair Loop.** Upon failure, the agent constructs a *Repair Prompt* that synthesizes the original intent Q, the erroneous query, and the error context. This feedback drives the LLM to generate a corrected query, resolving semantic ambiguities undetectable via static analysis.

The agent iterates through this static-dynamic loop until a valid result is obtained or the retry budget is exhausted. The detailed error categorization and specific correction algorithms are provided in Appendix A.3.

A detailed theoretical complexity analysis of the above three modules is presented in Appendix D.

5 Experiments

In this section, we evaluate the effectiveness and robustness of our **Adaptive Text2GQL** framework across diverse LLMs and Graph Query Languages (Cypher, Gremlin, SPARQL). Furthermore, we conduct ablation studies to validate the contributions of our three core paradigms proposed in Sec. 4.

5.1 Experimental Setup

5.1.1 Datasets

To ensure comprehensive evaluation, we utilize two distinct datasets: a curated **Multi-Language Movie Benchmark** and the official **Neo4j Text2Cypher 2024v1 Benchmark** (Neo4j-Text2Cypher). The former, built on a real-world knowledge graph (approx. 170 entities, 230 relationships) with 240 linguistically diverse QA pairs, features manual alignment across Cypher, Gremlin, and SPARQL to rigorously test cross-dialect adaptability. The latter serves as a large-scale industrial standard to verify the framework's robustness and generalization capabilities in complex, high-volume Cypher generation scenarios.

5.1.2 Graph Query Languages and LLMs

We employ standard database engines for execution: *Neo4j 5.24* (Cypher), *TinkerPop 3.7* (Gremlin), and *Jena Fuseki 5.6* (SPARQL). For generation, we test three distinct LLMs: **GPT-4o** and **DeepSeek V3.2** (state-of-the-art general reasoning

Table 1: Main Results: Generation Performance across GQLs and LLMs with Best Results Underlined

GQL	LLM	Our Method (Full)			w/o AEGC			Neo4j-Text2Cypher			R^3-NL2GQL			AutoKGQA		
		EM	GAM	EX	EM	GAM	EX	EM	GAM	EX	EM	GAM	EX	EM	GAM	EX
Cypher	GPT-4o	87.73	**94.12**	**66.25**	**90.10**	93.29	62.50	79.49	87.54	44.45	83.86	91.33	50.28	76.94	84.00	52.56
	DeepSeek	84.29	92.25	49.38	84.15	91.89	49.38	83.29	88.90	49.38	81.44	85.76	43.13	62.96	74.90	16.07
	Qwen3	83.35	89.22	59.71	82.40	88.19	57.21	79.55	85.12	44.32	75.03	83.73	41.41	74.26	81.13	47.20
Gremlin	GPT-4o	72.05	72.91	**62.50**	71.19	72.44	58.75	66.36	68.52	52.22	71.46	70.91	35.56	69.93	73.43	58.87
	DeepSeek	71.92	**75.72**	61.50	**72.24**	75.53	60.25	66.65	71.06	41.25	70.91	72.08	45.00	67.45	75.28	58.88
	Qwen3	66.67	69.39	43.75	65.99	67.52	37.50	64.52	67.48	34.56	67.27	70.72	32.22	67.54	68.11	44.60
SPARQL	GPT-4o	70.70	88.23	**52.40**	70.90	**88.69**	51.15	62.32	87.38	13.65	**76.33**	86.84	44.23	37.03	64.92	12.50
	DeepSeek	68.06	87.49	30.52	68.33	87.26	30.52	66.39	84.61	23.65	71.95	88.42	35.00	32.72	55.08	10.71
	Qwen3	61.93	83.93	14.90	61.96	84.26	14.90	51.86	72.01	7.50	70.48	85.42	30.00	29.92	50.29	14.88

models), and **Qwen3-Coder** (specialized for structured code generation). This selection assesses the framework's adaptability across both foundational and domain-specific model families.

5.1.3 Baselines and Evaluation Metrics

We conduct a comprehensive evaluation across three GQLs, benchmarking against competitive methods including Neo4j-Text2Cypher (Ozsoy et al., 2025), R^3-NL2GQL (Zhou et al., 2024), and AutoKGQA (Avila et al., 2024). Since these baselines originally lack native support for cross-language generation, we manually adapted their pipelines to enable a unified evaluation across Cypher, Gremlin, and SPARQL.

To assess performance from syntactic and semantic perspectives, we employ three metrics:

- **Exact Match (EM):** Measures strict string-level equality between the generated and reference queries after normalization.
- **GQL-AST Match (GAM):** A structural metric that builds Abstract Syntax Trees (AST) for both queries and compares their structures, capturing syntactic similarity while ignoring formatting differences.
- **Execution Accuracy (EX):** Assesses semantic correctness (Hong et al., 2025) by executing both queries on the graph databases and comparing the result sets.

5.1.4 Implementation Details

All experiments were conducted on an NVIDIA RTX 3090 GPU environment. The optimal hyperparameter configurations utilized in our framework were rigorously derived from the parameter sensitivity analysis discussed in Section 5.6.

- **Representation:** We utilize the lightweight **BAAI/bge-small-en-v1.5** (Xiao et al., 2024) for embedding computation.
- **Structural Twig Linking:** In the retrieval phase, schema element matches are weighted at $\gamma = 0.8$ to prioritize topological correctness over lexical overlap.
- **Evolutionary In-Context Learning:** For shot selection, we balance structural relevance and semantic similarity equally ($\alpha = \beta = 1.0$). The time decay factor is strictly calibrated to $\lambda = 0.001$, and the utility update learning rate is optimized to $\eta = 0.3$ to ensure stable evolutionary adaptation.
- **Adversarial Correction:** The agent's maximum repair budget is set to $k_{\max} = 2$ to prevent error amplification.

5.2 Cross-Language Generation Results

Our framework establishes a new state-of-the-art across all GQLs (Table 1), peaking at **66.25% EX** and **94.12% GAM** in Cypher. It also consistently empowers open-weight models (e.g., Qwen3) to substantially outperform existing baselines. Notably, while R^3-NL2GQL achieves slightly higher EM in SPARQL, this is an artifact of its skeleton-based approach that **pre-populates extensive boilerplate prefixes** identical to the ground truth. Despite this surface-level similarity, our method significantly leads in execution accuracy (**52.40%** vs. 44.23%), proving superior fidelity in generating valid query logic. Furthermore, the *Adversarial Execution-Guided Correction (AEGC)* module universally enhances robustness, notably yielding a **+6.25%** EX gain for Qwen3 in Gremlin by effectively repairing structural hallucinations.

Generalization. To further validate the robustness, we evaluated on the mixed-domain **Neo4j Text2Cypher 2024v1** benchmark. Despite the absence of an execution environment, our method achieves **81.34% GAM** and **75.34% EM**, confirming that our framework generalizes effectively to diverse schemas even in static generation scenarios.

5.3 Ablation Study of Structural Twig Linking

We validate **Structural Twig Linking** via the *Twig Hit Rate*—defined as the proportion of retrieved structural priors explicitly present in the ground truth query. As shown in Table 2, our strategy achieves >60% precision across all GQLs. Notably, SPARQL peaks at 80.33% due to its rigid triple patterns, whereas Gremlin trails (63.98%) given its procedural flexibility. These results confirm that identified "Twigs" effectively anchor NL intent to topological skeletons.

Table 2: Twig Hit Rate (%) across GQLs

Metric	Cypher	Gremlin	SPARQL
Twig Hit Rate	77.00	63.98	80.33

5.4 Ablation Study of Evolutionary In-Context Learning

We evaluate **Evolutionary ICL** against static few or zero shot (Table 3). Our strategy consistently dominates, particularly in Execution Accuracy. Notably, standard *Few-shot* suffers from negative transfer in complex GQLs, underperforming *Zero-shot* in Gremlin (EM drops $\sim$15.6%) and SPARQL. We attribute this drop to structural noise from irrelevant static examples. Similar phenomena have also been reported in previous studies (Li et al., 2024b,a). By evolving the shot repository based on syntactic utility, our method filters this noise, ensuring only execution-verified patterns guide generation. This result validates our ability to solve the cold-start problem and boost fidelity without curated datasets.

Table 3: Evolutionary ICL vs. Baselines (GPT-4o)

GQL	Strategy	EM (%)	GAM (%)	EX (%)
Cypher	**Evolutionary**	**87.73**	**94.12**	**66.25**
	Few-shot	81.58	88.67	56.25
	Zero-shot	85.70	89.61	48.75
Gremlin	**Evolutionary**	**72.05**	**72.91**	**62.50**
	Few-shot	53.58	55.09	38.66
	Zero-shot	69.23	64.89	42.22
SPARQL	**Evolutionary**	**70.70**	**88.23**	**52.40**
	Few-shot	56.22	75.23	24.80
	Zero-shot	66.93	84.63	22.79

5.5 Ablation Study of Adversarial Correction

We evaluate robustness via controlled noise injection (Table 4). While noisy inputs retain high textual overlap (EM >91%) yet fail execution (EX drops to $\sim$31-45%), AEGC demonstrates remarkable resilience. It successfully bridges this fidelity gap, restoring EX to **>83%** across all languages and achieving near-perfect syntactic validity (GAM reaches 99.20% in SPARQL). This result confirms that our *Dynamic Verifier* targets and repairs structural hallucinations that evade static textual matching.

Table 4: AEGC Robustness under Noise Injection

GQL	Input with Noise			After AEGC		
	EM	GAM	EX	EM	GAM	EX
Cypher	91.31	81.17	31.25	96.98	94.65	87.83
Gremlin	91.60	64.81	36.25	91.80	92.85	85.00
SPARQL	95.21	87.03	45.00	94.29	99.20	83.75

5.6 Hyperparameter Sensitivity and Robustness

To validate the robustness of our framework and ensure optimal configuration, we conduct a comprehensive sensitivity analysis on key hyperparameters. We evaluate the impact of the utility update learning rate (η), the shot scoring parameters (α, β, λ), and the twig linking weight (γ).

Our evaluations reveal that the framework maintains a broad plateau of stable performance across a wide range of parameter variations, demonstrating strong algorithmic robustness. For instance, the Execution Accuracy (EX) remains highly stable at **62.50%** across a broad moderate range of twig linking weights ($\gamma \in [0.2, 0.8]$). Furthermore, through this rigorous grid search, we successfully identified the optimal configurations that synergistically maximize the overall metrics, pushing the peak EX to **66.25%** (with $\eta = 0.3$). Detailed experimental setups, complete performance tables, and in-depth discussions for all evaluated hyperparameters are provided in Appendix E.

5.7 Robustness to Query and Schema Complexity

To gain deeper insights into our framework's capabilities under varying conditions, we conduct a granular analysis across different complexity levels. We categorize instances based on two dimensions: query topological complexity (proxied by target GQL length) and database schema scale.

Our evaluations demonstrate that our framework maintains remarkable structural stability across both metrics. As query difficulty increases to the "Hard" level, while strict Execution Accuracy natu-

rally declines, our approach sustains a robust GAM of **86.80%** on the custom dataset, gracefully degrading on the hard dataset where standard baselines completely fail (**0%** EX). Furthermore, when scaling to massive database schemas (exceeding 10,000 characters), our framework entirely prevents the "lost-in-the-middle" degradation typical of standard LLM generation. Even on the hardest schema subsets, our framework maintains a highly stable GAM of **81.23%** and achieves a peak EM of **70.11%**. Detailed experimental setups, performance tables across complexity bins, and further analyses are provided in Appendix D.

5.8 Cost and Latency Analysis

We analyze the trade-off between generation quality and computational overhead. While our framework introduces latency through embedding-based retrieval and iterative correction, this cost is amortized as the *Adversarial Execution-Guided Correction (AEGC)* loop is only triggered upon error detection. Statistically, the average inference latency increases from $\approx$**3.80s** (Naive baseline) to $\approx$**8.58s** (Ours). While it represents a $2.3\times$ increase, the absolute latency remains within a manageable range for analytical applications, especially given the substantial gains in execution accuracy.

Regarding token consumption, our efficient schema pruning keeps the average input prompt at merely **325.6 tokens**. The *AEGC* module exhibits a selective activation rate of **15.2%**, ensuring that expensive repair iterations are reserved only for complex failures. A comprehensive breakdown of runtime latency and token efficiency is provided in **Appendix G**.

6 Related Work

Text-to-SQL. Text-to-SQL is a mature domain (Kanburoğlu and Tek, 2024) anchored by large-scale benchmarks like WikiSQL (Hwang et al., 2019) and Spider variants (Yu et al., 2018; Lei et al., 2024), which demand advanced multi-table reasoning. Methodologies generally split into two paradigms: *fine-tuning* language models on massive code corpora (e.g., CodeS (Li et al., 2024a), SQL-LLaMA (Roziere et al., 2023)) to internalize database syntax, and *In-Context Learning* utilizing schema retrieval (Zhang et al., 2023; Guo et al., 2023) to dynamically provide structural context during the generation process. As the field advances to tackle increasingly complex programmatic queries—such as procedural extensions of SQL (Zhang et al., 2025)—autonomous execution-guided verification (Fan et al., 2025; Li and Xie, 2024) has become standard practice for iteratively ensuring query fidelity via runtime feedback.

Text-to-GQL. Compared to relational databases, graph query generation faces unique topological complexities and severe data scarcity challenges. Early studies predominantly relied on task-specific fine-tuning via GraphQL definitions (Ganesan et al., 2024; Quiña-Mera et al., 2023) or synthetic Cypher generation pipelines (Tiwari et al., 2025) to compensate for the lack of parallel training data. After identifying schema linking as a critical bottleneck in navigating dense graph topologies (Zhou et al., 2024; Deutsch et al., 2022), recent research has shifted towards enhancing LLM robustness through execution refinement (Liang et al., 2024b), latent pattern exploitation (Liu et al., 2024), and structured domain alignment (Liang et al., 2024a). Furthermore, the field has recently expanded into multi-turn conversational interactions (Liang et al., 2025) and the development of specialized Text2SPARQL models tailored for complex knowledge graph applications (Avila et al., 2024; Ben Amor et al., 2025). However, existing research remains largely "single language centric", limiting its applicability across the increasingly fragmented graph database ecosystem. We directly address this gap by proposing a unified framework designed for robust, cross-language generalization across diverse GQL standards.

7 Conclusion

In this work, we proposed **Adaptive Text2GQL**, a unified, training-free framework addressing topological heterogeneity and structural hallucinations in cross-language graph query generation. Our approach synergizes three core mechanisms: **Structural Twig Linking** for scalable schema grounding, **Evolutionary In-Context Learning** for robust cold-start adaptation, and **Adversarial Execution-Guided Correction** for iterative query repair. Extensive experiments demonstrate state-of-the-art execution accuracy and structural alignment across Cypher, Gremlin, and SPARQL. By effectively preventing "lost-in-the-middle" degradation on massive schemas, our framework delivers a highly scalable natural language interface for the fragmented graph database ecosystem.

8 Limitations

Despite its robustness, our framework exhibits limitations regarding runtime dependency and computational overhead. First, the *Adversarial Execution-Guided Correction (AEGC)* module relies on database feedback; in static "pure generation" settings without connectivity, the system cannot verify dynamic constraints, leading to a performance discount compared to execution-enabled scenarios. Second, the embedding-based *Structural Twig Linking* introduces pre-processing latency, while the iterative nature of self-correction increases token consumption and operational costs. Future work will aim to mitigate these issues by exploring quasi-offline verification mechanisms to reduce strict database dependencies and investigating more lightweight, cost-effective architectures to optimize both retrieval efficiency and inference overhead.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China (No. 62372264 and No. 92467203) and Sina Weibo Corp. Chaokun Wang is the corresponding author.

References

Caio Viktor S Avila, Vânia MP Vidal, Wellington Franco, and Marco A Casanova. 2024. Experiments with text-to-sparql based on chatgpt. In *2024 IEEE 18th international conference on semantic computing (ICSC)*, pages 277–284. IEEE.

Mehdi Ben Amor, Alexis Strappazzon, Michael Granitzer, Elöd Egyed-Zsigmond, and Jelena Mitrović. 2025. Instruct-to-sparql: A text-to-sparql dataset for training sparql agents. In *Proceedings of the 2025 ACM SIGIR Conference on Human Information Interaction and Retrieval*, pages 390–395.

Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, and 1 others. 2022. Graph pattern matching in gql and sql/pgq. In *Proceedings of the 2022 International Conference on Management of Data*, pages 2246–2258.

Yuankai Fan, Tonghui Ren, Can Huang, Zhenying He, and X Sean Wang. 2025. Grounding natural language to sql translation with data-based self-explanations. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*, pages 29–42. IEEE.

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*, pages 1433–1445.

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*, pages 8657–8660.

Chunxi Guo, Zhiliang Tian, Jintao Tang, Pancheng Wang, Zhihua Wen, Kang Yang, and Ting Wang. 2023. Prompting gpt-3.5 for text-to-sql with desemanticization and skeleton retrieval. In *Pacific Rim International Conference on Artificial Intelligence*, pages 262–274. Springer.

Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, and 1 others. 2021. Knowledge graphs. *ACM Computing Surveys (Csur)*, 54(4):1–37.

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

Wonseok Hwang, Jinyeong Yim, Seunghyun Park, and Minjoon Seo. 2019. A comprehensive exploration on wikisql with table-aware word contextualization. *arXiv preprint arXiv:1902.01069*.

ISO/IEC JTC 1/SC 32. 2024. ISO/IEC 39075:2024 – Information Technology – Database Languages – GQL. Technical report, International Organization for Standardization (ISO).

Ali Buğra Kanburoğlu and Faik Boray Tek. 2024. Text-to-sql: A methodical review of challenges and models. *Turkish Journal of Electrical Engineering and Computer Sciences*, 32(3):403–419.

Lyndon S Kennedy, Apostol Natsev, and Shih-Fu Chang. 2005. Automatic discovery of query-class-dependent models for multimodal search. In *Proceedings of the 13th annual ACM international conference on multimedia*, pages 882–891.

Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo, Hongcheng Gao, Wenjing Hu, Pengcheng Yin, and 1 others. 2024. Spider 2.0: Evaluating language models on real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*.

Haoyang Li, Jing Zhang, Hanbing Liu, Ju Fan, Xiaokang Zhang, Jun Zhu, Renjie Wei, Hongyan Pan, Cuiping Li, and Hong Chen. 2024a. Codes: Towards building open-source language models for text-to-sql. *Proceedings of the ACM on Management of Data*, 2(3):1–28.

Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024b. Llm-r2: A large language model enhanced rule-based rewrite system for boosting query efficiency. *Proceedings of the VLDB Endowment*, 18(1):53–65.

Zhenwen Li and Tao Xie. 2024. Using llm to select the right sql query from candidates. *CoRR*.

Yuanyuan Liang, Lei Pan, Tingyu Xie, Yunshi Lan, and Weining Qian. 2025. Multi-turn natural language to graph query language translation. *arXiv preprint arXiv:2508.01871*.

Yuanyuan Liang, Keren Tan, Tingyu Xie, Wenbiao Tao, Siyuan Wang, Yunshi Lan, and Weining Qian. 2024a. Aligning large language models to a domain-specific graph database for nl2gql. In *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, pages 1367–1377.

Yuanyuan Liang, Tingyu Xie, Gan Peng, Zihao Huang, Yunshi Lan, and Weining Qian. 2024b. Nat-nl2gql: A novel multi-agent framework for translating natural language to graph query language. *CoRR*.

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. ISSN*, 2150:8097.

Neo4j-Text2Cypher. 2024. https://huggingface.co/datasets/neo4j/text2cypher-2024v1. (last updated on 2025-03).

Makbule Gulcin Ozsoy, Leila Messallem, Jon Besga, and Gianandrea Minneci. 2025. Text2cypher: Bridging natural language and graph databases. In *Proceedings of the Workshop on Generative AI and Knowledge Graphs (GenAIK)*, pages 100–108.

Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and complexity of sparql. *ACM Transactions on Database Systems (TODS)*, 34(3):1–45.

Antonio Quiña-Mera, Pablo Fernandez, José María García, and Antonio Ruiz-Cortés. 2023. Graphql: A systematic mapping study. *ACM computing surveys*, 55(10):1–35.

Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th symposium on database programming languages*, pages 1–10.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Shashank Sheshar Singh, Samya Muhuri, Shivansh Mishra, Divya Srivastava, Harish Kumar Shakya, and Neeraj Kumar. 2024. Social network analysis: A survey on process, tools, and application. *ACM computing surveys*, 56(8):1–39.

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.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Shitao Xiao, Zheng Liu, Peitian Zhang, Niklas Muennighoff, Defu Lian, and Jian-Yun Nie. 2024. C-pack: Packed resources for general chinese embeddings. In *Proceedings of the 47th international ACM SIGIR conference on research and development in information retrieval*, pages 641–649.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, and 1 others. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887*.

Hanchong Zhang, Ruisheng Cao, Lu Chen, Hongshen Xu, and Kai Yu. 2023. Act-sql: In-context learning for text-to-sql with automatically-generated chain-of-thought. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 3501–3532.

Hang Zhang, Chaokun Wang, Hongwei Li, Cheng Wu, Songyao Wang, Yabin Liu, Gengyuan Shi, and Ziyang Liu. 2025. Plforge: Enhancing language models for natural language to procedural extensions of sql. *Proceedings of the ACM on Management of Data*, 3(6):1–28.

Leqi Zheng, Chaokun Wang, Canzhi Chen, Jiajun Zhang, Cheng Wu, Zixin Song, Shannan Yan, Ziyang Liu, and Hongwei Li. 2025a. Lagcl4rec: When llms activate interactions potential in graph contrastive learning for recommendation. In *Findings of the Association for Computational Linguistics: EMNLP 2025, Suzhou, China, November 4-9, 2025*, pages 1163–1184. Association for Computational Linguistics.

Leqi Zheng, Chaokun Wang, Zixin Song, Cheng Wu, Shannan Yan, Jiajun Zhang, and Ziyang Liu. 2025b. Negative feedback really matters: Signed dual-channel graph contrastive learning framework for recommendation. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*.

Leqi Zheng, Jiajun Zhang, Canzhi Chen, Chaokun Wang, Hongwei Li, Yuying Li, Yaoxin Mao, Shannan Yan, Zixin Song, Zhiyuan Feng, Zhaolu Kang, Zirong Chen, Hang Zhang, Qiang Liu, Liang Wang,

and Ziyang Liu. 2026. What should i cite? a rag benchmark for academic citation prediction. In *Proceedings of the ACM Web Conference 2026*, WWW '26, page 1852–1863, New York, NY, USA. Association for Computing Machinery.

Yuhang Zhou, Yu He, Siyu Tian, Yuchen Ni, Zhangyue Yin, Xiang Liu, Chuanjun Ji, Sen Liu, Xipeng Qiu, Guangnan Ye, and 1 others. 2024. r3-nl2gql: A model coordination and knowledge graph alignment approach for nl2gql. In *Findings of the association for computational linguistics: EMNLP 2024*, pages 13679–13692.

A Algorithmic Details

In this appendix, we provide the detailed algorithmic procedures for the core modules of our Adaptive Text2GQL framework, specifically the retrieval logic for Structural Twig Linking and the dual-phase refinement loop for Adversarial Execution-Guided Correction.

A.1 Structural Twig Retrieval Algorithm

As discussed in Sec. 4.1, the Structural Twig Linking module bridges the gap between linear text and graph topology. Algorithm 1 formally describes the ranking and selection process. It scores candidate twigs $\mathcal{H}_{\text{full}}$ based on a hybrid metric combining:

- **Schema Consistency (score_s):** The proportion of schema elements in the twig that are successfully grounded in the mapping table $\mathcal{M}$.
- **Lexical Resonance (score_l):** The IDF-weighted overlap between the twig's lexical tags and the filtered user query tokens.

Algorithm 1 Structural Twig Retrieval Algorithm

Require: Full Twig Set $\mathcal{H}_{\text{full}}$, Mapping $\mathcal{M}$, Tokens $\mathbf{T}_{\text{filtered}}$, Budget K
Ensure: Selected Twigs $\mathcal{H}_{\text{topK}}$

1: $\mathcal{H}_{\text{scored}} \leftarrow \emptyset$
2: **for** $h \in \mathcal{H}_{\text{full}}$ **do**
3: $\quad T_s \leftarrow \text{ExtractSchemaTags}(h)$
4: $\quad T_l \leftarrow \text{RetrieveLexicalTags}(h)$
5: $\quad \triangleright$ Compute Schema Consistency
6: $\quad \text{score}_s \leftarrow |T_s \cap \text{Values}(\mathcal{M})| / |T_s|$
7: $\quad \triangleright$ Compute Lexical Resonance
8: $\quad \text{score}_l \leftarrow \sum_{w \in T_l} \text{IDF}(w) \cdot \mathbb{I}(w \in \mathbf{T}_{\text{filtered}})$
9: $\quad \text{FinalScore} \leftarrow \gamma \cdot \text{score}_s + (1-\gamma) \cdot \text{score}_l$
10: $\quad \mathcal{H}_{\text{scored}}.\text{add}((h, \text{FinalScore}))$
11: **end for**
12: $\mathcal{H}_{\text{topK}} \leftarrow \text{Top}_K(\mathcal{H}_{\text{scored}}, \text{key=score})$
13: **return** $\mathcal{H}_{\text{topK}}$

A.2 Evolutionary Shot Selection Details

In Sec. 4.2, we introduced the utility-driven selection mechanism for Cold-Start adaptation. Algorithm 2 details the scoring logic used to retrieve the most effective structural demonstrations from the self-growing repository $\mathcal{L}$.

The selection relies on a composite score balancing two factors:

- **Topological Relevance (sim):** Calculated via the Jaccard similarity of schema elements (nodes and edges) between the new query's grounding $\mathcal{M}_{\text{new}}$ and the historical shot's context $\mathcal{M}_i$.
- **Syntactic Utility (utility):** Derived from the shot's historical execution confidence ρ_i, decayed by its age τ_i to prioritize recent adaptations.

Algorithm 2 Evolutionary Shot Selection

Require: Query Q_{new}, Mapping $\mathcal{M}_{\text{new}}$, Repository $\mathcal{L}$, Budget K
Ensure: Selected Context $\mathcal{E}_{\text{shots}}$

1: $\mathbf{Candidates} \leftarrow \emptyset$
2: **for** $e_i \in \mathcal{L}$ **do**
3: $\quad \triangleright$ Compute Topological Relevance
4: $\quad \mathcal{S}_{\text{new}} \leftarrow \text{ExtractSchema}(\mathcal{M}_{\text{new}})$
5: $\quad \mathcal{S}_i \leftarrow \text{ExtractSchema}(\mathcal{M}_i)$
6: $\quad \text{sim} \leftarrow \frac{|\mathcal{S}_i \cap \mathcal{S}_{\text{new}}|}{|\mathcal{S}_i \cup \mathcal{S}_{\text{new}}|}$
7: $\quad \triangleright$ Compute Syntactic Utility
8: $\quad \text{utility} \leftarrow \rho_i \cdot e^{-\lambda \tau_i}$
9: $\quad \text{score} \leftarrow \alpha \cdot \text{sim} + \beta \cdot \text{utility}$
10: $\quad \mathbf{Candidates}.\text{add}((e_i, \text{score}))$
11: **end for**
12: $\mathcal{E}_{\text{shots}} \leftarrow \text{TopK}(\mathbf{Candidates}, K)$
13: **return** $\mathcal{E}_{\text{shots}}$

A.3 Adversarial Execution-Guided Correction Details

Supplementing Sec. 4.3, we detail the workflow of the Correction Agent.

Error Categorization. We classify feedback into three structured contexts:

1. **Syntactic Errors:** Violations of GQL grammar (e.g., missing colons in Cypher) captured by the database parser.

2. **Semantic Errors:** Queries that are syntactically valid but yield empty results or violate schema logic (e.g., referencing a non-existent property).

3. **Runtime Errors:** Execution failures due to resource constraints (e.g., timeout).

Correction Algorithm. Algorithm 3 outlines the AEGC process, illustrating the synergy between the Static Critic (using adversarial priors) and the Dynamic Verifier.

Algorithm 3 Adversarial Execution-Guided Correction

Require: Initial Query G_{gen}, Question Q, Schema $\mathcal{S}$, Adversarial Priors $\mathcal{E}_{\text{adv}}$
Ensure: Valid GQL G_{valid} or Failure

1: $G_{\text{curr}} \leftarrow G_{\text{gen}}$
2: $k \leftarrow 0$
3: **while** $k < k_{\max}$ **do**
4:   **if** MatchAdversarialPattern($G_{\text{curr}}, \mathcal{E}_{\text{adv}}$) **then**
5:     $G_{\text{curr}} \leftarrow$ Agent$_{\text{refine}}$(G_{curr}, "Static violation")
6:     **continue**
7:   **end if**
8:   $(result, error) \leftarrow$ ExecuteOnDB(G_{curr})
9:   **if** $error =$ null **and** Validate($result$) **then**
10:     **return** G_{curr}
11:   **end if**
12:   $E \leftarrow$ ExtractErrorContext($error$)
13:   $prompt \leftarrow$ ConstructRepairPrompt($Q, \mathcal{S}, G_{\text{curr}}, E$)
14:   $G_{\text{curr}} \leftarrow$ Agent$_{\text{refine}}$($prompt$)
15:   $k \leftarrow k + 1$
16: **end while**
17: **return Failure**

B Template Repository Construction Details

Our **Structural Twig Linking** module systematically generates query hints by introspecting the graph schema. Based on the logic defined in our pattern generator, we categorize templates into four distinct groups: *Node Patterns*, *Relationship Patterns*, *Complex Patterns*, and *Aggregation Patterns*.

Below, we provide the specific template libraries for Cypher, Gremlin, and SPARQL. In each table, the **Abstract** row displays the structural skeleton (where □ denotes a slot for schema/value injection), while the **Example** row shows a concrete instantiation with its corresponding Schema Tags (T_s) and Natural Language Tags (T_{nl}).

B.1 Cypher Template Repository

Cypher relies on declarative ASCII-art pattern matching. Table 5 details the core patterns.

B.2 Gremlin Template Repository

Gremlin employs procedural traversals. Table 6 illustrates how traversal steps are linked.

B.3 SPARQL Template Repository

SPARQL utilizes RDF triple patterns within a `WHERE` clause. Table 7 shows the triple structures.

C Static Error Base Details

To support the **Static Critic** module, we constructed a predefined Error Base ($\mathcal{B}_{\text{err}}$) containing common hallucination patterns observed in LLM-generated graph queries. Table 8 lists representative examples of these heuristic rules.

The error base categorizes rules into two types:

- **Syntax Regex:** Regular expressions designed to catch formatting errors (e.g., missing delimiters, unclosed brackets).
- **Logic Anti-Pattern:** Structural checks for queries that are syntactically parsable but semantically invalid (e.g., disconnected components or aggregating without grouping).

D Complexity Analysis

In this section, we provide a theoretical analysis of the computational complexity for the two primary retrieval-based modules in our framework: **Structural Twig Linking** and **Evolutionary In-Context Learning**. We omit the complexity of LLM inference as it is constant given a fixed model and token limit.

D.1 Structural Twig Linking Complexity

This module integrates *Semantic Grounding* and *Topological Instantiation*. Let:

- $n = |\mathbf{T}_{\text{filtered}}|$: Number of filtered tokens in the query.
- $m = |\mathcal{S}|$: Total number of schema elements (nodes + edges).
- d: Dimension of the embedding vectors.
- $N_{\text{twig}} = |\mathcal{H}_{\text{full}}|$: Total number of instantiated candidate twigs.

Semantic Grounding. Identifying atomic schema elements requires embedding projection and pairwise similarity computation.

Table 5: Cypher: Abstract Structural Templates and Instantiated Examples

Category	Type	Pattern (Abstract / Example)	Tags (T_s / T_{nl})
Node	*Basic*	**Abs:** `(:□)` **Ex:** `(:Person)`	– T_s: **Person** T_{nl}: find, show, all
	Property	**Abs:** `(:□ {□: □})` **Ex:** `(:Movie {title: $val})`	– T_s: **Movie, title** T_{nl}: named, called, filter
Rel.	*Directed*	**Abs:** `(:□)-[:□]->(:□)` **Ex:** `(:Person)-[:DIRECTED]->(:Movie)`	– T_s: **DIRECTED** T_{nl}: director, made
	Path	**Abs:** `(:□)-[:□*1..3]->(:□)` **Ex:** `(:Person)-[:FOLLOWS*1..3]->(:Person)`	– T_s: **FOLLOWS** T_{nl}: path, indirect, connect
Complex	*Chain*	**Abs:** `(:□)-[:□]->(:□)-[:□]->(:□)` **Ex:** `(:P)-[:KNOWS]->(:P)-[:KNOWS]->(:P)`	– T_s: **KNOWS** T_{nl}: friend of friend, chain
	Star	**Abs:** (c:□)-[:□]-(n_1), (c)-[:□]-(n_2) **Ex:** `(m:Movie)<-[:ACT]-(p1),(m)<-[:DIR]-(p2)`	– T_s: **ACT, DIR** T_{nl}: center, multiple, cast
Agg.	*Count*	**Abs:** `MATCH (n:□) RETURN count(n)` **Ex:** `MATCH (n:City) RETURN count(n)`	– T_s: **City** T_{nl}: how many, total, quantity

Table 6: Gremlin: Abstract Structural Templates and Instantiated Examples

Category	Type	Pattern (Abstract / Example)	Tags (T_s / T_{nl})
Node	*Basic*	**Abs:** `g.V().hasLabel('□')` **Ex:** `g.V().hasLabel('Movie')`	– T_s: **Movie** T_{nl}: find, list
	Filter	**Abs:** `...hasLabel('□').has('□', □)` **Ex:** `...hasLabel('Person').has('age', gt(20))`	– T_s: **age** T_{nl}: older than, filter
Rel.	*Out*	**Abs:** `...out('□').hasLabel('□')` **Ex:** `...out('ACTED_IN').hasLabel('Movie')`	– T_s: **ACTED_IN** T_{nl}: acted in, role
	Bidir.	**Abs:** `...both('□')` **Ex:** `...both('FOLLOWS')`	– T_s: **FOLLOWS** T_{nl}: mutual, relationship
Complex	*Chain*	**Abs:** `...out('□').in('□')` **Ex:** `...out('PRODUCED').in('DIRECTED')`	– T_s: **PRODUCED**, **DIR.** T_{nl}: via, colleague
	Star	**Abs:** `...project('a','b').by(out('□'))...` **Ex:** `...by(out('ACTED')).by(out('DIRECTED'))`	– T_s: **ACTED**, **DIR.** T_{nl}: various, hub
Agg.	*Func*	**Abs:** `...values('□').mean()` **Ex:** `...values('rating').mean()`	– T_s: **rating** T_{nl}: average, typical

Table 7: SPARQL: Abstract Structural Templates and Instantiated Examples

Category	Type	Pattern (Abstract / Example)	Tags (T_s / T_{nl})
Node	*Type*	**Abs:** `?s a :□` **Ex:** `?s a :City`	– T_s: **City** T_{nl}: find, location
	Prop.	**Abs:** `?s a :□; :□ □ .` **Ex:** `?s a :Person; :bornIn "USA" .`	– T_s: **bornIn** T_{nl}: from, filter
Rel.	*Triple*	**Abs:** `?s :□ ?o . ?o a :□` **Ex:** `?s :WROTE ?o . ?o a :Book`	– T_s: **WROTE**, **Book** T_{nl}: author, wrote
	Path	**Abs:** `?s :□* ?o` **Ex:** `?s :knows* ?o`	– T_s: **knows** T_{nl}: connection, path
Complex	*Chain*	**Abs:** `?a :□ ?b . ?b :□ ?c` **Ex:** `?a :spouse ?b . ?b :mother ?c`	– T_s: **spouse**, **mother** T_{nl}: relative, via
	Star	**Abs:** `?c :□ ?n1; :□ ?n2` **Ex:** `?m :DIRECTED ?p; :PRODUCED ?c`	– T_s: **DIR.**, **PROD.** T_{nl}: center, details
Agg.	*Func*	**Abs:** `SELECT (AVG(?v)) WHERE {?s :□ ?v}` **Ex:** `SELECT (AVG(?v)) ... {?s :price ?v}`	– T_s: **price** T_{nl}: average cost, mean

Table 8: Representative Examples from the Static Error Base ($\mathcal{B}_{\text{err}}$)

GQL	Type	Error Pattern / Regex Signature	Description & Corrective Action
Cypher	Regex	`\(\w+\s+[A-Z]\w+\)`	**Missing Colon:** LLMs often forget the colon separator (e.g., `(n Person)`). *Fix:* Insert colon → `(n:Person)`.
	Regex	`ORDER BY\s+.*\s+RETURN`	**Clause Order:** `ORDER BY` cannot precede `RETURN`. *Fix:* Move `ORDER BY` after `RETURN`.
	Anti-Pattern	Disconnected Components	**Cartesian Product:** Variables in `MATCH` are not connected. *Fix:* Add relationship or `WHERE` clause constraint.
Gremlin	Regex	`\.out\(['"]\w+['"]\)\.in`	**Invalid Chain:** Direct `.out().in` without node definition. *Fix:* Ensure traversal logic or use `.both()`.
	Regex	`select\(([^)]+)\)\.by\(\)`	**Empty By-Modulator:** `.by()` requires arguments for selection. *Fix:* Inject property key → `.by('name')`.
	Anti-Pattern	SQL Keywords	**Hallucination:** Usage of `SELECT, WHERE, FROM` in Gremlin. *Fix:* Replace with `.select(), .has(), g.V()`.
SPARQL	Regex	`[^}]\s*$\s*`	**Missing Triple Terminator:** Triples must end with a dot (.). *Fix:* Append dot (.) to the line end.
	Regex	`FILTER\s*\(\?[a-zA-Z]+\s*=\s*".*"\)`	**String Comparison:** Using = for strings is risky in RDF. *Fix:* Replace with `REGEX()` or `STRSTARTS()`.
	Anti-Pattern	Unbound Variable	**Select Error:** Selecting a variable `?x` not present in `WHERE`. *Fix:* Remove `?x` or add defining triple.

- **Embedding:** Computing embeddings for tokens and schema elements takes $O((n+m)\cdot d)$.
- **Similarity:** Calculating the cosine similarity matrix between all tokens and schema elements requires $O(n\cdot m\cdot d)$ operations.

Thus, the grounding phase complexity is dominated by $O(n\cdot m\cdot d)$.

Topological Instantiation & Retrieval. This phase involves generating twigs and scoring them.

- **Instantiation:** Generating N_{twig} candidates via template filling is linear with respect to the output size, $O(N_{\text{twig}})$.
- **Scoring:** For each twig, we compute:
  1. *Schema Consistency:* Checking intersection with the mapping table $\mathcal{M}$ takes $O(|T_s|)$. In the worst case, $|T_s|\approx m$, so this is $O(m)$.
  2. *Lexical Resonance:* Checking overlap with query tokens takes $O(n)$ (assuming hash-based lookups).

  Therefore, scoring all twigs takes $O(N_{\text{twig}}\cdot(m+n))$.
- **Selection:** Selecting the top-K twigs using a heap takes $O(N_{\text{twig}}\log K)$.

Overall Complexity: The total time complexity for Structural Twig Linking is:

$$O(n\cdot m\cdot d + N_{\text{twig}}\cdot(m+n)).$$

Since N_{twig} can be controlled via template constraints and d is a constant, the process remains efficient for real-world graph schemas.

D.2 Evolutionary In-Context Learning Complexity

We analyze the maintenance and retrieval costs for the self-growing shot repository. Let:

- $L=|\mathcal{L}|$: Current size of the shot repository.
- $\bar{S}$: Average number of schema elements per shot.
- S_{new}: Number of schema elements in the new query.

Shot Selection. For a new query, we compute the relevance score for every shot in $\mathcal{L}$.

- **Topological Relevance:** Computing Jaccard similarity between the query schema and shot schema takes $O(S_{\text{new}}+\bar{S})$. Across the repository, this is $O(L\cdot(S_{\text{new}}+\bar{S}))$.
- **Utility & Sorting:** Computing the decayed utility score is $O(1)$ per shot. Selecting top-K shots takes $O(L\log K)$.

Repository Evolution.

- **Update:** Incrementing age τ and updating confidence ρ for all shots is $O(L)$.
- **Eviction:** Finding the minimum utility shot for eviction is $O(L)$ (or $O(1)$ if a min-heap is main-

tained, though re-weighting requires $O(L)$).

Overall Complexity: The complexity per inference step is dominated by the schema similarity computation:

$$O(L \cdot (S_{\text{new}} + \bar{S})).$$

Given that the repository capacity $|\mathcal{L}|_{\max}$ is a fixed hyperparameter (e.g., 100-500), this module introduces negligible latency compared to the LLM generation time.

E Detailed Parameter Sensitivity Analysis

This section provides the comprehensive results of the hyperparameter sensitivity analysis mentioned in Section 5.6. We aim to evaluate the impact of individual parameters and demonstrate the framework's robustness.

Utility Update Learning Rate (η). The learning rate η controls how rapidly the utility scores of historical shots adapt based on their execution feedback. As shown in Table 9, a moderate learning rate ($\eta \in \{0.3, 0.5\}$) yields optimal performance, peaking at 66.25% EX. A lower rate ($\eta = 0.1$) delays evolutionary adaptation, while higher rates ($\eta \geq 0.7$) overly penalize historical shots, leading to undesirable performance degradation.

Table 9: Sensitivity Analysis of Utility Update Learning Rate (η)

η	**EX (%)**	**EM (%)**	**GAM (%)**
0.1	62.50	86.80	93.05
0.3	**66.25**	**87.73**	**94.12**
0.5	66.25	87.21	93.85
0.7	58.75	85.74	91.95
0.9	58.75	86.12	92.20

Shot Scoring Parameters (α, β, λ). Table 10 details the performance across varying combinations of weights for structural similarity (α), semantic similarity (β), and the temporal decay factor (λ). The results explicitly validate that assigning full importance to structural matching ($\alpha = 1.0$) is critical. Combining this with balanced semantic matching ($\beta = 1.0$) and a gentle temporal decay ($\lambda = 0.001$) achieves the optimal synergy for maintaining a high-quality Evolutionary ICL repository.

Twig Linking Weight (γ). Table 11 evaluates the structural Twig Linking scoring function. The framework exhibits a broad plateau of stability,

Table 10: Sensitivity Analysis of Shot Scoring Parameters

α	β	λ	**EX (%)**	**EM (%)**	**GAM (%)**
1.0	1.0	0.1	59.50	84.20	91.10
1.0	1.0	0.01	62.50	86.80	93.05
1.0	**1.0**	**0.001**	**63.75**	**86.63**	**93.65**
1.0	1.0	0.0001	61.25	85.90	92.50
1.0	0.5	0.001	62.50	86.44	92.85
0.5	1.0	0.001	57.21	84.80	90.80

maintaining a solid 62.50% EX across a wide moderate range ($\gamma \in [0.2, 0.8]$). Performance drops are only observed at extreme bounds ($\gamma = 0.0$ or $\gamma = 1.0$), proving the overall robustness of the structural matching mechanism.

Table 11: Sensitivity Analysis of Twig Linking Weight (γ)

γ	**EX (%)**	**EM (%)**	**GAM (%)**
0.0	60.63	86.14	91.50
0.2	62.50	86.80	93.05
0.4	62.50	86.87	92.90
0.6	62.50	86.50	93.10
0.8	62.50	86.45	92.70
1.0	60.63	86.37	91.80

F Query and Schema Complexity Analysis

This section provides the comprehensive results of the Query and Schema complexity analysis mentioned in Section 5.7. To thoroughly evaluate the module efficacy under varying conditions, we assess performance across different levels of query topological complexity and schema scale.

F.1 Query Complexity Analysis

Since standard graph database datasets lack official difficulty splits, we categorize the queries into *Easy*, *Medium*, and *Hard* bins based on the character length of the target GQL. We use strict thresholds of 80 and 120 characters as a proxy for topological complexity. This granular evaluation is conducted across both our Custom Dataset (Table 12) and the multi-domain Neo4j Dataset (Table 13).

Analysis. As demonstrated in Tables 12 and 13, our framework effectively handles the Text2GQL task for Easy and Medium queries, achieving high Execution Accuracy and robust structural alignment. Crucially, as query topological complexity increases, our method maintains remarkable stability in structural metrics.

Table 12: Performance across Query Complexity Bins (Custom Dataset)

Difficulty	Length Threshold	EM (%)	GAM (%)	EX (%)
Easy	< 80 chars	94.62	96.58	85.71
Medium	$80 - 120$ chars	88.63	92.92	59.46
Hard	> 120 chars	66.62	86.80	26.67

Table 13: Performance across Query Complexity Bins (Neo4j Dataset)

Difficulty	Length Threshold	EM (%)	GAM (%)
Easy	< 80 chars	71.37	85.31
Medium	$80 - 120$ chars	68.36	80.46
Hard	> 120 chars	65.55	80.50

While strict Execution Accuracy (EX) naturally declines on Hard queries, it is notable that all baseline models completely fail (achieving 0% EX) under the exact same splits, highlighting a universal challenge in Text2GQL. Despite these execution bottlenecks, our framework sustains a robust structural alignment, preserving a GAM of 86.80% on the Custom dataset and 80.50% on the Neo4j dataset. This gracefully degraded performance confirms that our Structural Twig Linking and Evolutionary ICL mechanisms effectively capture complex graph topologies even when strict execution fails. *Note: EX is omitted for the Neo4j Dataset because it consists of highly heterogeneous schemas without the corresponding backend database instances required for dynamic query execution; thus, evaluation strictly focuses on structural metrics.*

F.2 Schema Complexity Analysis

To evaluate our framework's scalability against varying database sizes, we partitioned the multi-domain Neo4j benchmark into three balanced complexity bins based on the character length of the target database schema, which ranges from 72 to over 10,400 characters.

Table 14: Metrics across Schema Complexity Bins (Neo4j Benchmark)

Difficulty	Schema Length Threshold	EM (%)	GAM (%)
Easy	< 274 chars	62.89	81.79
Medium	$274 - 2221$ chars	69.77	81.01
Hard	> 2221 chars	70.11	81.23

Analysis. As indicated in Table 14, our framework exhibits exceptional scalability and robustness against massively increasing schema complexities. Remarkably, even when operating on the Hardest subset—where the database schema encompasses extensive topologies and noise exceeding 2,221 characters—the GAM remains highly stable at approximately 81%. Furthermore, the Exact Match (EM) metric demonstrates a slight upward trend on larger schemas, peaking at 70.11%. This strongly validates that our Structural Twig Linking module acts as a highly effective semantic filter. It successfully pinpoints exact topological anchors within massive graph schemas, entirely preventing the catastrophic "lost-in-the-middle" structural degradation typically suffered by standard LLMs when processing lengthy contexts.

G Detailed Efficiency Analysis

In this section, we provide a granular breakdown of both inference latency and token consumption statistics.

G.1 Latency Comparison

Table 15 compares the average inference time between the Naive baseline (direct generation) and our framework across 80 independent runs. While our method introduces a necessary latency overhead (averaging ∼8.58s compared to the baseline's ∼3.80s) due to the retrieval process and iterative repair steps for complex queries, we maintain high efficiency through several system-level optimizations.

Specifically, we employ persistent database connectivity to eliminate redundant network handshakes, alongside offline schema pre-computation and local vector caching to transform runtime semantic grounding into a fast similarity search. These strategies effectively bypass heavy I/O and on-the-fly embedding overhead during inference.

Table 15: Average Inference Latency per Query (Seconds)

Language	Model	Naive (s)	Ours (s)	Factor
Cypher	GPT-4o	2.60	7.70	×3.0
	DeepSeek-V3.2	3.15	7.13	×2.3
	Qwen3-Coder	1.33	8.61	×6.5
Gremlin	GPT-4o	3.68	10.03	×2.7
	DeepSeek-V3.2	3.96	8.67	×2.2
	Qwen3-Coder	3.05	7.68	×2.5
SPARQL	GPT-4o	4.74	8.72	×1.8
	DeepSeek-V3.2	6.79	9.35	×1.4
	Qwen3-Coder	4.93	9.32	×1.9
Average		**3.80s**	**8.58s**	≈ ×2.3

G.2 Token Distribution Profile

Cost efficiency is a core design principle of our framework. Figure 3 illustrates the composition of the input prompt, which averages **325.6 tokens** per query.

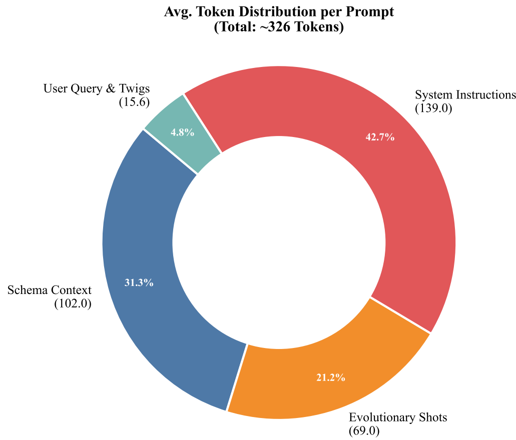

Figure 3: Average input token distribution. The *Compact Schema Representation* effectively limits the schema context to approx. 31% of the total usage, leaving ample room for system instructions and dynamic shots.

The distribution highlights the efficiency of our prompt construction strategy:

- **Compact Schema Representation:** We employ a concise linearization format for graph schema definitions. This optimized representation occupies only $\sim$102 tokens (31.3%), effectively minimizing redundancy while retaining essential topological information.
- **Dynamic Context:** The retrieval-augmented few-shot examples consume $\sim$69 tokens (21.2%), ensuring the model receives relevant guidance without inflating costs.
- **System Instructions:** The majority of the remaining context ($\sim$42.7%) is dedicated to rigid instruction adherence, ensuring syntactical correctness across different GQLs.