# ExGQL: Self-Improving Natural Language-to-Graph Query Generation via Experience-Guided Learning

**Penny Guo*** **Boxin Du*** **Houliang Zhou*** **Chaman Gupta***
**Yina Gu** **Rayssa Kullian** **Shen Jiang**
Amazon
{penghong,boxin,houliang,imchaman,yinagu,rayssam,shejiang}@amazon.com

## Abstract

Natural language-to-graph query generation (NL2GQL) requires schema-aware reasoning and multi-hop traversal planning, posing graph-specific challenges beyond the extensively studied Text-to-SQL setting. Existing NL2GQL systems typically solve each query in isolation, limiting their ability to learn from prior executions. We present ExGQL, an experience-guided framework for self-improving NL2GQL. ExGQL combines hierarchical memory, experience evolution, and capability synthesis to distill execution feedback into reusable reasoning strategies and convert validated strategies into executable capabilities. We further introduce a Failure-Directed Curriculum (FDC) for a separate practice phase, in which recurring failure patterns guide the selection of tasks that address the system's current weaknesses. We evaluate ExGQL on NL2GQL-Bench, a benchmark of 400 expert-annotated graph query tasks over a production-scale graph database. ExGQL achieves an average accuracy of 0.869 versus 0.662 for the strongest evaluated baseline (+31% relative improvement), while reducing average query-generation time by 28%. In a separate practice phase, FDC reaches comparable accuracy of 0.861 with 60% fewer practice episodes and 36% lower runtime than fixed-order practice. These results show that NL2GQL systems can improve by reusing execution-derived experience, while failure-directed practice makes experience acquisition more efficient.

## 1 Introduction

Graph databases are increasingly used to model complex relational systems in domains such as fraud detection, recommendation, cybersecurity, and enterprise knowledge management (Besta et al., 2023). Querying these systems, however, requires expertise in specialized graph query languages, including nGQL, openCypher, and Gremlin. These languages expose graph-specific constructs such as directional traversal, path constraints, variable scoping, and multi-hop composition, creating a substantial barrier for non-expert users. Natural-language interfaces to graph databases seek to bridge this gap by mapping user requests to executable graph queries (Liang et al., 2024a,b).

Natural language-to-graph query generation (NL2GQL) is related to the extensively studied Text-to-SQL problem. Recent advances in large language models (LLMs) have substantially improved Text-to-SQL through schema-aware prompting, retrieval augmentation, iterative refinement, and agent-based reasoning (Pourreza and Rafiei, 2023; Li et al., 2024; Wang et al., 2024a; Talaei et al., 2024). Graph query generation, however, presents a distinct reasoning challenge. Graph queries are traversal centric, often requiring multi-hop reasoning over heterogeneous entities and directed relationships. Graph query languages also remain less standardized than SQL, with database systems differing in syntax and execution semantics. Moreover, producing a correct query requires jointly reasoning about traversal direction, path composition, constraints, intermediate variables, and aggregation across multiple stages (Zhou et al., 2024). Consequently, NL2GQL is not merely code generation in a different syntax; it requires explicit reasoning over graph structure and execution semantics.

Despite recent progress, existing NL2GQL systems remain largely query-isolated: each request is solved from the model's current context, with limited support for revising and reusing knowledge derived from prior executions. Retrieval can surface examples or documentation, while multi-agent workflows can decompose generation and refinement within a request. However, these mechanisms are typically designed to support the current request rather than evolve execution-derived knowledge across requests. Consequently, errors such as

*Equal contribution.

incorrect schema grounding, traversal direction, or constraint propagation may recur despite having been corrected previously.

This limitation is particularly costly for NL2GQL because many requests share recurring schema-grounding, traversal, temporal, and repair patterns. Knowledge acquired while solving one request can therefore directly inform subsequent queries over the same or structurally similar schemas.

Recent work on Agentic Context Engineering (ACE) (Zhang et al., 2025) suggests that LLM agents can improve through structured context evolution rather than parameter updates. Motivated by this direction, we formulate NL2GQL as an experience-guided graph reasoning problem in which execution feedback is distilled into reusable knowledge for future query generation. We use experience to denote distilled, actionable guidance derived from prior executions, rather than raw interaction traces or retrieved examples. Such experience captures transferable strategies for schema grounding, traversal construction, and failure repair. The objective therefore extends beyond solving individual requests to improving NL2GQL performance over time.

Beyond learning from naturally occurring query interactions, ExGQL can bootstrap and strengthen its experience base through a separate practice phase. The effectiveness of this phase depends on how a finite practice budget is allocated. Fixed-order or uniform schedules may devote many episodes to skills the system already handles reliably, while persistent failure modes remain underexplored. Because the learning value of a task depends on the system's current competence, practice should prioritize tasks that expose unresolved capability gaps.

To this end, we propose ExGQL, an experience-guided framework for self-improving natural language-to-graph query generation. ExGQL combines hierarchical memory, experience evolution, and capability synthesis to distill execution feedback into reusable reasoning strategies and synthesize validated strategies into executable capabilities. It further introduces a Failure-Directed Curriculum that identifies recurring failure patterns and directs practice toward tasks that address them. Together, these mechanisms improve future query generation without updating model parameters.

Our contributions are threefold:

1. **Cross-query experience evolution.** We develop ExGQL, a self-improving NL2GQL framework that transfers execution-derived experience across queries and synthesizes validated reasoning strategies into executable capabilities, without model fine-tuning.

2. **Failure-directed practice.** We propose a Failure-Directed Curriculum (FDC) that identifies recurring failure patterns and selectively practices the tasks that most directly address them, improving practice efficiency by achieving comparable average accuracy with 60% fewer practice episodes while reducing the training runtime by 36%.

3. **Benchmark and evaluation.** We construct NL2GQL-Bench, a benchmark comprising 400 expert-annotated graph-query tasks over a production-scale NebulaGraph deployment. Under controlled conditions, ExGQL achieves 0.869 average accuracy versus 0.662 for the strongest evaluated baseline, a 31% relative improvement, while reducing reasoning cost by 28%.

## 2 Related Work

### 2.1 Natural Language Query Generation

Natural language query generation has been widely studied in Text-to-SQL, where LLM-based systems leverage decomposition, execution-guided refinement, and agentic reasoning to improve query correctness (Pourreza and Rafiei, 2023; Li et al., 2024; Wang et al., 2024a). In contrast, NL2GQL remains relatively underexplored, with existing approaches relying on schema grounding, fine-tuning, or multi-step generation pipelines (Zhou et al., 2024; Liang et al., 2024a,b). Despite their differences, both Text-to-SQL and NL2GQL systems typically solve queries independently without accumulating reusable experience across tasks.

### 2.2 Efficient example selection

Curriculum learning improves learning efficiency by scheduling training examples from easy to hard rather than sampling them uniformly (Bengio et al., 2009), and automated curricula replace fixed schedules with signals derived from the model's own learning progress (Graves et al., 2017). Active learning pursues a complementary objective, selecting the most informative examples to label under a limited budget (Settles, 2009), a principle

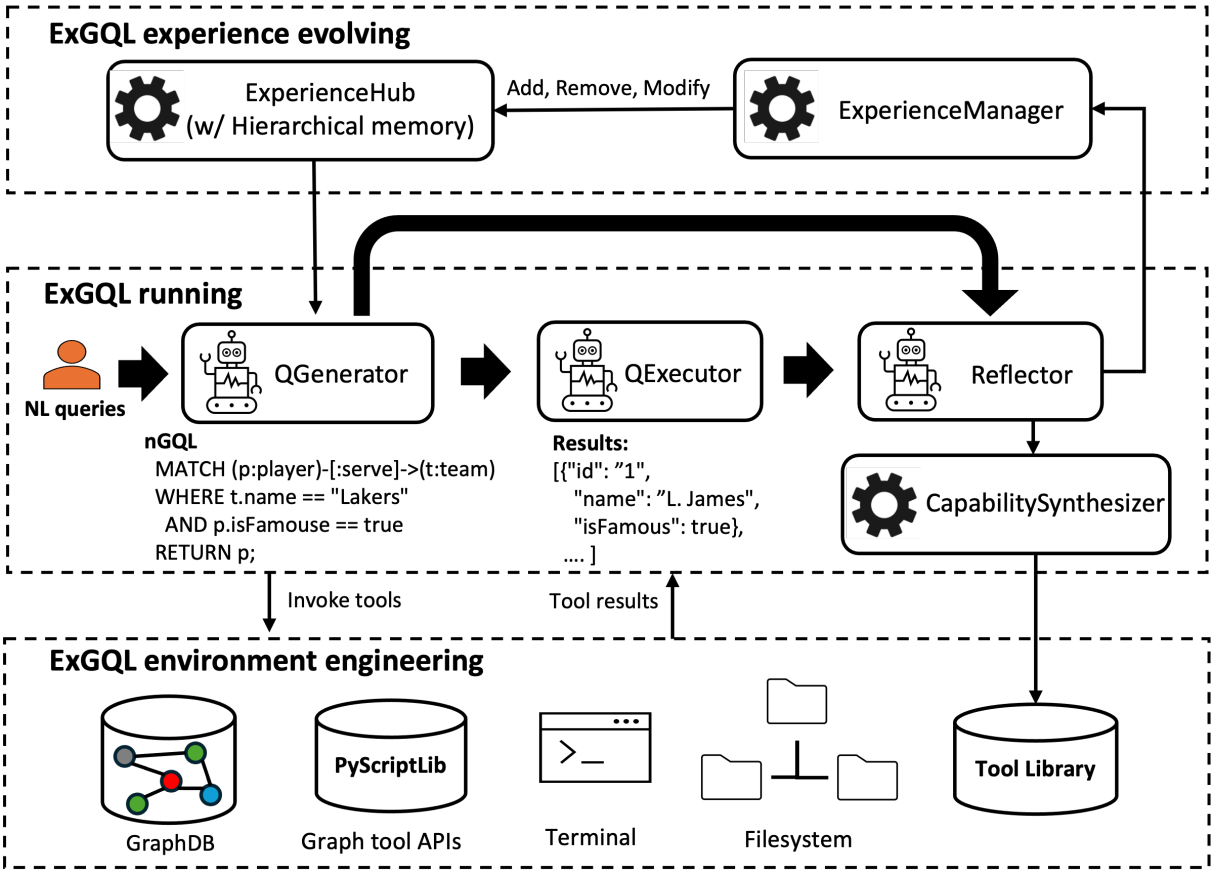



Figure 1: Overview of ExGQL. The agent retrieves relevant experience from hierarchical memory to generate executable graph queries, then distills execution feedback into reusable reasoning strategies and executable capabilities for future queries.

recently extended to large language models by selecting or ordering in-context demonstrations to improve few-shot performance without parameter updates (Zhang et al., 2022; Su et al., 2022; Margatina et al., 2023). These methods, however, target a model's parameters or its input demonstrations. ExGQL instead applies failure-directed selection to experience accumulation: the curriculum determines which tasks the agent practices, and each practiced task reshapes a persistent skillbook rather than a weight matrix or a prompt window. Unlike active learning, which selects examples to label, FDC selects examples to practice, using its own diagnosed failures rather than label uncertainty as the informativeness signal.

### 2.3 Self-Improving LLM Systems

Recent work has shown that LLMs can improve through experience accumulation rather than parameter updates, using reflection, memory, and evolving context to reuse knowledge across tasks (Shinn et al., 2023; Suzgun et al., 2026; Zhang et al., 2025; Wang et al., 2024b; Xu et al., 2025; Agrawal et al., 2026). However, these approaches primarily target general reasoning and agentic workflows. ExGQL extends this paradigm to graph query generation, where execution correctness, schema constraints, and multi-hop reasoning require domain-specific experience representations and execution-guided knowledge evolution.

## 3 Problem Formulation

We formulate natural language to graph task solving as an experience-evolving reasoning process. Given a natural language request $\mathcal{X}$ and graph schema $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P})$, the objective is to generate an action sequence that solves the user task through graph queries, tool invocations, and intermediate reasoning. At iteration $t$, the action of the agent can be represented as:

$$\hat{\mathcal{A}}_t = f(\mathcal{X}, \mathcal{G}, \mathcal{C}_t, \mathcal{T}_t), \tag{1}$$

where $\mathcal{C}_t$ denotes accumulated experience and $\mathcal{T}_t$ denotes the current reasoning trajectory. Execution feedback continuously updates $\mathcal{C}_t$, enabling

continual improvement without parameter updates.

NL-to-nGQL translation is a primary instantiation of this framework, where the objective is to generate executable nGQL statements that correctly answer a natural language query.

## 4 Methodology

### 4.1 ExGQL Overview

Figure 1 illustrates the overall architecture of ExGQL. The framework consists of two tightly coupled execution loops.

The reasoning loop solves the current task by retrieving relevant knowledge from the hierarchical memory, dynamically allocating reasoning budget, generating graph queries, and invoking executable capabilities when appropriate.

The experience evolution loop continuously improves the system by reflecting on completed executions, distilling reusable reasoning strategies, updating the hierarchical memory, and promoting mature reasoning into executable capabilities.

The remainder of this section describes the core mechanisms of ExGQL: hierarchical memory, experience evolution, capability synthesis, and failure-directed curriculum.

### 4.2 Hierarchical Memory

A central design principle of ExGQL is that different forms of knowledge should not be represented or maintained in the same way. Instead of storing all accumulated information within a single memory, ExGQL organizes knowledge into a hierarchical memory composed of three complementary layers.

The first layer, the Template Cache, stores frequently occurring graph query patterns that can be instantiated with minimal reasoning. These templates provide efficient retrieval for routine graph analysis tasks and avoid repeatedly reconstructing common query structures.

The second layer, the Experience Hub, serves as the primary reasoning memory of ExGQL. Rather than storing raw execution traces, it maintains reusable reasoning experience distilled from previous executions, including traversal strategies, schema-grounding heuristics, temporal reasoning patterns, debugging strategies, and execution heuristics. Unlike conventional retrieval memories that simply retrieve similar examples, the Experience Hub stores generalized reasoning knowledge that transfers across different graph reasoning tasks.

The third layer, the Knowledge Base, stores validated long-term knowledge accumulated through repeated interactions. It contains graduated playbook learnings that have consistently demonstrated utility and have been promoted out of the in-context playbook to reduce token consumption, as well as domain-specific expertise, such as graph behavioral patterns, analytical heuristics, and other validated knowledge that supports complex graph reasoning tasks. Unlike static documentation repositories, the Knowledge Base itself evolves through continual execution and validation.

During reasoning, ExGQL retrieves information from all three layers simultaneously. The Template Cache supplies reusable query patterns, the Experience Hub contributes evolving reasoning knowledge, and the Knowledge Base provides validated long-term knowledge accumulated through continual learning.

### 4.3 Adaptive Reasoning

Existing NL2GQL systems typically process every request using a fixed reasoning pipeline. However, graph reasoning workloads exhibit substantial variation in complexity. Some tasks correspond to well-known graph patterns and require little reasoning, while others involve multi-hop traversal, temporal reasoning, graph analytics, or iterative planning.

ExGQL therefore adopts an adaptive reasoning strategy that allocates computation and reasoning step budget according to task complexity. Upon receiving a natural-language request, the planner estimates the reasoning complexity based on the query structure, required graph operations, retrieved experience, and execution dependencies. Straightforward queries primarily reuse template cache or executable capabilities (e.g., executable functions or tools), whereas more challenging tasks invoke deeper planning, iterative reasoning, intermediate graph execution, and additional retrieval from the Experience Hub.

Unlike conventional retrieval-augmented generation, adaptive reasoning jointly determines what knowledge should be retrieved, how much reasoning should be performed, and which executable capabilities should be invoked. This dynamic allocation substantially reduces unnecessary retrieval, while preserving the ability to solve complex graph reasoning tasks.

### 4.4 Experience Evolution

Following each execution, ExGQL reflects on the complete reasoning process to identify knowledge that generalizes beyond the current task. The reflection module analyzes execution trajectories, generated graph queries, intermediate reasoning steps, execution outcomes, and tool interactions. Instead of preserving complete execution histories, ExGQL extracts reusable reasoning knowledge that is likely to benefit future graph reasoning tasks.

Candidate experiences include traversal strategies, schema-grounding heuristics, temporal reasoning patterns, debugging heuristics, execution ordering principles, and task decomposition strategies. Each experience is associated with supporting execution evidence and confidence estimates. As additional executions accumulate, similar experiences are merged, conflicting experiences are reconciled, and outdated experiences are gradually deprecated. Consequently, the Experience Hub continuously evolves from a collection of isolated examples into a repository of reusable graph reasoning knowledge.

### 4.5 Capability Synthesis

While reusable reasoning experience significantly improves graph reasoning, not all accumulated experience should remain as textual format. As the Experience Hub grows, ExGQL periodically identifies reasoning strategies that have become sufficiently mature and repetitive to justify procedural abstraction.

Instead of repeatedly asking the language model to interpret identical reasoning patterns, ExGQL consolidates similar reasoning trajectories, extracts their common procedural structure, parameterizes variable components, and synthesizes reusable executable capabilities. Candidate capabilities are validated through repeated execution before being registered as callable graph reasoning operators. These are reusable tools that directly solve subtasks of the natural language query, automating the processes of repeated tasks handled from scratch if they are not available.

Unlike manually engineered agent tools, these capabilities emerge directly from accumulated execution experience. Future graph reasoning tasks invoke these capabilities directly rather than reconstructing the underlying reasoning process, substantially reducing reasoning cost while improving execution consistency. Capability synthesis therefore represents the transition from learning only reusable reasoning strategies to learning reusable graph reasoning operators.

### 4.6 Failure-Directed Curriculum

The mechanisms above determine how ExGQL learns from a single task. We further adopt a failure-directed curriculum to determine which tasks it practices, when the agent is trained on a set of examples for building the Experience Hub and capabilities. Rather than practicing the full training set in a fixed order, ExGQL learns in rounds under a practice budget $B$: at each round it diagnoses its current failures on a held-out probe $\mathcal{D}$ and practices only the examples that target them.

Let $s(x; M) \in [0, 1]$ be the normalized score of task $x$ under memory state $M$, and let round $r$ hold memory $M_r$. The algorithm scores a disposable copy of $M_r$ on $\mathcal{D}$, leaving $M_r$ unchanged, and takes the failing tasks $\mathcal{F} = \{x \in \mathcal{D} : s(x; M_r) < 1\}$ (i.e., any task not solved perfectly). Failures with an identical structural key including category, subtype, schema-path signature, complexity, output shape, error class, and execution route form a cluster $c$, scored by a deficit-weighted priority

$$\pi(c) = \sum_{x \in c} \big(1 - s(x; M_r)\big), \tag{2}$$

so clusters that fail more often, or more severely, rank higher. Let $\mathcal{P} \subseteq \mathcal{Q}$ denote the tasks already practiced in previous rounds. From the remaining pool $\mathcal{P} \setminus \mathcal{Q}$, ExGQL selects a batch that matches the highest-priority clusters, preferring candidates that share the most structural fields with a cluster before broadening to the same category.

The agent practices the batch with learning enabled, and the round repeats until the budget $B$ is exhausted, no probe task fails, or the diagnostic macro plateaus. The diagnostic macro $g(M) = \frac{1}{8}\sum_{j=1}^{8} \phi_j(M)$ averages the eight evaluation metrics $\phi_j$ computed on $\mathcal{D}$, summarizing overall competence in a single score. ExGQL then returns the checkpoint

$$M^\star = \arg\max_{0 \le i \le r} \; g(M_i), \tag{3}$$

breaking ties toward fewer practiced examples and lower cost. Checkpoint selection reads only the diagnostic probe $\mathcal{D}$, never the held-out evaluation set, and the agent, prompts, and tools are held fixed across rounds. So the curriculum is the only variable that differs from standard full-set practice.

### 4.7 Continual Experience-Guided Learning

Putting everything together, the previous components are integrated through a continual execution-and-learning loop. During training, the Failure-Directed Curriculum would allow the agent to select the tasks to focus on from the training examples. For each graph reasoning task, ExGQL retrieves relevant knowledge from the hierarchical memory, dynamically allocates reasoning effort, generates and executes graph operations, and receives grounded execution feedback.

After execution, the system reflects on the complete reasoning process, refines the Experience Hub with newly distilled reasoning strategies, promotes validated knowledge into the Knowledge Base when appropriate, and periodically synthesizes mature reasoning strategies into executable capabilities. Over time, ExGQL continuously evolves its hierarchical memory, allowing future graph reasoning tasks to benefit from accumulated reasoning experience without parameter updates.

## 5 Experiments

### 5.1 Dataset

We evaluate all methods on a production-scale heterogeneous property graph used for e-commerce fraud investigation. The graph snapshot used in our experiments contains approximately 40.6 billion vertices and 98.9 billion edges, spanning 22 vertex types and 24 directed edge types.

Because the underlying graph is part of a proprietary production system, we cannot release its full schema, raw contents, or domain-specific identifiers for confidentiality and security reasons. We instead report aggregate structural statistics and provide sanitized task descriptions and examples that illustrate the benchmark design without exposing sensitive information.

We construct NL2GQL-Bench, an internal benchmark of 400 natural language-to-nGQL tasks, divided into 200 practice tasks and 200 held-out test tasks. The benchmark covers five categories: Search, Reachability, $k$-hop Subgraph, Property Aggregation, and Pattern Matching. These categories span single-entity and property retrieval, multi-hop traversal, aggregation, and structural pattern matching. Each task is created through a human-in-the-loop annotation process: domain experts author natural language queries, generate candidate nGQL queries with LLM assistance (Claude Opus 4.6), and manually verify the execution results. Detailed task definitions and examples are provided in Appendix A.1.

**Practice and test protocol.** The practice tasks are used to accumulate experience and synthesize executable capabilities; the underlying language-model parameters remain fixed. During held-out evaluation, both the hierarchical memory and capability set are frozen, and feedback from test tasks is not retained or used for subsequent predictions.

### 5.2 Implementation Details

All methods are evaluated under a controlled experimental setting to isolate the impact of system design. ExGQL and all baselines use the same foundation language model (Claude Opus 4.6) and have identical access to the graph schema, graph-query execution capabilities, tool interfaces, and execution feedback. No method performs supervised fine-tuning or parameter updates. ExGQL and its ablated variants are implemented within the same Bedrock-based agentic framework, whereas the ReAct baselines follow a standard ReAct implementation. Although their orchestration differs, all methods operate under functionally equivalent conditions for query generation and execution.

**Baselines.** The current NL2GQL literature remains relatively limited and heterogeneous. Existing approaches differ substantially in graph dialects, training requirements, execution environments, and released implementations, and many rely on domain-specific supervised fine-tuning or proprietary datasets. These differences make direct comparison on a shared production graph impractical. We therefore compare ExGQL against representative reasoning baselines together with carefully designed ablated variants that isolate the effects of hierarchical memory, experience evolution, and capability synthesis.

**Baseline 1: ExGQL w/o Hierarchical Memory.** This baseline removes access to the hierarchical memory while preserving the overall multi-agent architecture. The agent performs iterative reasoning and execution using the same planner, executor, and reflection pipeline as ExGQL, but cannot retrieve or reuse accumulated experience across queries.

**Baseline 2: ExGQL w/o Hierarchical Memory & Capability Synthesis.** This baseline further removes learned executable capabilities, forcing the agent to reconstruct all reasoning through the language model for every query. Comparing this

baseline with ExGQL isolates the combined contribution of reusable experience and synthesized capabilities.

**Baseline 3: Code Agent.** A ReAct-style single-agent baseline that performs iterative reasoning with tool invocation and execution feedback within an individual query. The agent has no persistent memory and does not accumulate knowledge across queries.

**Baseline 4: Few-shot Code Agent.** A Code Agent augmented with five representative query examples per benchmark category, including natural-language questions, ground-truth nGQL queries, and execution results. This baseline evaluates whether static in-context examples can substitute for continually accumulated reasoning experience.

### 5.3 Results

**Metrics** We evaluate ExGQL and the four baselines across five representative query types in the benchmark dataset. We conduct a ground-truth accuracy evaluation by comparing agent outputs against reference answers of the benchmark dataset. For Search, we calculate the accuracy by extracting the primary numeric answer and check whether it matches the reference within 5% relative error. For aggregation, we compute accuracy by extracting all metric values and compute the fraction that match reference values. For k-hop traversal, we measure node recall, edge recall, and hop completeness (whether the correct number of hops are traversed by the nGQL queries) against the reference subgraph. For reachability, we check whether the agent reaches the correct conclusion (whether two entities are connected vs. not connected), and whether the reported shortest path length matches. For pattern matching, we measure precision as the fraction of NL-specified conditions satisfied by the matched entities from the agent output.

**Accuracy Analysis** Table 1 presents the ground-truth accuracy comparison. ExGQL achieves the best performance on search (0.850), aggregation (0.631), reachability conclusion (0.957), and pattern matching precision (0.796). The hierarchical memory contributes most to search (+0.100) and pattern matching (+0.082), where accumulated nGQL syntax rules and traversal patterns prevent repeated errors. Removing tools has modest impact on most categories, confirming that the primary performance driver is the structured experience context rather than specialized tool functions. The Stateless Code Agent achieves competitive k-hop node recall (0.856 with few-shot) by performing exhaustive traversals within its unlimited step budget, but struggles on aggregation (0.304) and pattern matching (0.340) where multi-step reasoning and constraint composition are required. Introducing the FDC preserves the overall accuracy of ExGQL while sharply reducing the practice required. Trained on only a targeted subset of practice tasks, the curriculum variant attains an average accuracy of 0.861. Overall, ExGQL achieves an average accuracy of 0.869 across all evaluation metrics, outperforming the strongest baseline (0.662 for the few-shot Code Agent) by 31%, with the largest gains on aggregation (+0.327) and pattern matching (+0.456).

**Execution Efficiency** Table 2 reports the average number of agent steps and wall-clock time per query category. Each agent or tool invocation is counted as one step. ExGQL achieves the lowest step count (14.24 average) and fastest execution (273.8s average) across all categories. The most dramatic efficiency gain is on k-hop subgraph retrieval, where ExGQL's unified subgraph retrieval query completes in 6.7 steps (17s) compared to 17.7 steps (348s) for the tool-free baseline—a 20× speedup enabled by the orchestrator-side subgraph fast-path. For pattern matching, all methods require substantial step budgets (36–41 steps), but ExGQL's structured decomposition and tool reuse reduce wall-clock time by 12% compared to the no-memory variant.

**Practice Efficiency** The accuracy reported above is retained at a fraction of the practice effort. Holding the agent, prompts, and tools fixed and varying only how practice examples are selected, the ExGQL with FDC reaches the accuracy reported while using only 80 practice examples–40% of the full practice set, and 36% of the practice cost. By concentrating practice on diagnosed weakness rather than consuming the practice set indiscriminately, ExGQL now accumulates a comparably effective hierarchical memory from substantially fewer executions, each of which invokes the full reasoning and execution pipeline.

## 6 Conclusion

Translating natural language into executable graph queries remains challenging due to schema heterogeneity, multi-hop reasoning, and strict exe-

| Category | ExGQL (with FDC) | ExGQL (w/o FDC) | ExGQL (w/o mem) | ExGQL (w/o mem & capability) | ReAct Code Agent | Few Shot Code Agent |
|---|---|---|---|---|---|---|
| Search (accuracy) | 0.835 | **0.850** | 0.750 | 0.750 | 0.400 | 0.400 |
| Aggregation (accuracy) | **0.673** | 0.631 | 0.603 | 0.592 | 0.239 | 0.304 |
| k-hop: node recall | **0.856** | **0.843** | 0.714 | 0.514 | 0.796 | 0.856 |
| k-hop: edge recall | 0.800 | **0.875** | 0.693 | 0.447 | 0.593 | 0.653 |
| k-hop: hop completeness | **1.000** | **1.000** | 0.829 | 0.854 | **1.000** | **1.000** |
| Reachability (conclusion) | **0.972** | 0.957 | 0.913 | 0.870 | 0.739 | 0.739 |
| Reachability (path) | **1.000** | **1.000** | **1.000** | **1.000** | **1.000** | **1.000** |
| Pattern matching (precision) | 0.750 | **0.796** | 0.714 | 0.778 | 0.308 | 0.340 |

Table 1: Ground-truth accuracy evaluation. ExGQL achieves the highest scores across all categories except k-hop node recall and hop completeness, where the ReAct Code Agent achieves competitive performance due to exhaustive traversal within its unlimited step budget. ExGQL (with FDC) is trained under a failure-directed curriculum on a targeted subset of practice tasks and remains non-inferior to the full-set model (average 0.861 vs. 0.869), trading small per-category losses on k-hop edge recall and pattern matching for gains on aggregation and reachability.

| Category | ExGQL | | ExGQL (w/o memory) | | ExGQL (w/o memory & capability) | |
|---|---|---|---|---|---|---|
| | Steps | Time (s) | Steps | Time (s) | Steps | Time (s) |
| Search | **9.0** | **56** | 12.2 | 84 | 6.5 | 126 |
| Aggregation | **13.8** | **313** | 17.3 | 333 | 16.9 | 319 |
| k-hop Traversal | **6.7** | **17** | 6.8 | 175 | 17.7 | 348 |
| Reachability | **5.2** | **118** | 5.7 | 172 | 6.4 | 183 |
| Pattern Matching | **36.5** | **865** | 40.8 | 982 | 39.5 | 940 |
| AVERAGE | **14.24** | **274** | 16.56 | 349 | 17.4 | 383 |

Table 2: Execution efficiency comparison. ExGQL requires fewer steps and less time across all categories. The k-hop speedup (17s vs. 348s) is attributed to the learnings of the unified subgraph query in the hierarchical memory, while pattern matching savings come from reusable predicate tools.

cution requirements. We presented ExGQL, an experience-enhanced framework that reformulates NL-to-nGQL translation as an experience accumulation process. By combining compositional skill planning, a persistent hierarchical memory, failure-directed curriculum, and execution-guided learning, ExGQL enables continual improvement through the reuse of distilled experience, leading to improved execution accuracy and robustness on complex graph-query tasks. Future work includes extending ExGQL to additional graph query languages (e.g., OpenCypher and Gremlin) and exploring cross-database generalization.

## Limitations

ExGQL is evaluated exclusively on nGQL over a single production-scale graph schema. While the framework's compositional planning and experience accumulation mechanisms are designed to be language-agnostic, transferability to other graph query languages (e.g., OpenCypher, Gremlin) and different graph schema remains unverified. The hand-curated synonym dictionaries, hub-node exclusion lists, and domain-specific optimizations would require adaptation for new domains. Additionally, NL2GQL-Bench contains 200 test queries authored by domain experts, which may not fully capture the ambiguity of real end-user queries or exhaustively cover all graph operation types. The failure-directed curriculum targets only failure modes that appear in its held-out probe set, and its efficiency gains are demonstrated under a single practice-budget schedule that may not transfer to other benchmarks or graph schema.

The framework relies on a large language model for all agent roles, introducing substantial inference cost that scales with query complexity and refinement iterations. Long-term hierarchical memory scalability beyond the 200-query training phase is also not evaluated; as the repository grows, retrieval quality may degrade without more sophisticated pruning or hierarchical organization strategies.

## References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, and 1 others. 2026. Gepa: Reflec-

tive prompt evolution can outperform reinforcement learning. In International Conference on Learning Representations, volume 2026, pages 8479–8565.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In Proceedings of the 26th Annual International Conference on Machine Learning (ICML), pages 41–48.

Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. 2023. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. ACM Computing Surveys, 56(2):1–40.

Alex Graves, Marc G. Bellemare, Jacob Menick, Rémi Munos, and Koray Kavukcuoglu. 2017. Automated curriculum learning for neural networks. In Proceedings of the 34th International Conference on Machine Learning (ICML).

Chaofan Li, Yingxia Shao, Yawen Li, and Zheng Liu. 2024. Sea-sql: Semantic-enhanced text-to-sql with adaptive refinement. arXiv preprint arXiv:2408.04919.

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. arXiv preprint arXiv:2412.10434.

Katerina Margatina, Timo Schick, Nikolaos Aletras, and Jane Dwivedi-Yu. 2023. Active learning principles for in-context learning with large language models. In Findings of the Association for Computational Linguistics: EMNLP 2023.

Mohammadreza Pourreza and Davood Rafiei. 2023. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. Advances in neural information processing systems, 36:36339–36348.

Burr Settles. 2009. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, Department of Computer Sciences.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36.

Hongjin Su, Jungo Kasai, Chen Henry Wu, Weijia Shi, Tianlu Wang, Jiayi Xin, Rui Zhang, Mari Ostendorf, Luke Zettlemoyer, Noah A Smith, and Tao Yu. 2022. Selective annotation makes language models better few-shot learners. In The Eleventh International Conference on Learning Representations.

Mirac Suzgun, Mert Yuksekgonul, Federico Bianchi, Dan Jurafsky, and James Zou. 2026. Dynamic cheatsheet: Test-time learning with adaptive memory. In Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers), pages 7080–7106.

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

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2024a. Mac-sql: A multi-agent collaborative framework for text-to-sql. arXiv preprint arXiv:2312.11242.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. 2024b. Agent workflow memory. arXiv preprint arXiv:2409.07429.

Wujiang Xu and 1 others. 2025. A-mem: Agentic memory for llm agents. arXiv preprint arXiv:2502.12110.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. 2025. Agentic context engineering: Evolving contexts for self-improving language models. arXiv preprint arXiv:2510.04618.

Yiming Zhang, Shi Feng, and Chenhao Tan. 2022. Active example selection for in-context learning. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing (EMNLP).

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.

# Appendix

## A Detailed Method Specification

### A.1 Benchmark Dataset Details

We summarize the details of the benchmark dataset in Table 3.

Table 3: Benchmark task categories with definitions and example queries.

| **Category** | **Definition** | **Example NL Query** |
|---|---|---|
| Search | Single-entity lookup that retrieves a specific scalar value (e.g., maximum, count, interval) for one customer within a time window. | *"What is the maximum order value of the purchases for customer* `<Customer_ID>` *over the last 3 months?"* |
| Aggregation | Multi-hop traversal from a seed entity (address, credit card, or customer) to collect associated customers, then compute aggregate statistics. | *"For the normalized address* `<Address_ID>`*, get the average account age, average concession value, and average concession interval of all customers who shipped orders to this address."* |
| $k$-Hop Subgraph | Starting from a center node, extract the $k$-hop neighborhood (3 or 4 hops) following schema-defined edge directions, with at most 5 nodes per hop per type, skipping hub nodes ($\geq$10,000 reverse degree). | *"Starting from* `<Customer_ID>`*, get its 3-hop outgoing subgraph. Max 5 nodes per hop per type. Skip hub nodes and report them."* |
| Reachability | Determine whether two customer nodes are connected within a bounded number of hops (6 or 8), and if so, return the shortest path(s) between them. | *"Check whether* `<Customer_ID>` *and* `<Customer_ID>` *are connected within 8 hops. If so, output the connecting path."* |
| Pattern Matching | Identify groups of customers exhibiting a specified behavioral pattern (e.g., high reversal rate, shared payment instruments). | *"Find 10 groups of >5 customers sharing relations from [IP, EmailAddress, Address], over the last month."* |