GraphScout: Empowering Large Language Models with Intrinsic Exploration Ability for Agentic Graph Reasoning

Anonymous Author(s)

Abstract

Knowledge graphs provide structured and reliable information for many real-world applications, motivating increasing interest in combining large language models (LLMs) with graph-based retrieval to improve factual grounding. Recent Graph-based Retrieval-Augmented Generation (GraphRAG) methods therefore introduce iterative interaction between LLMs and knowledge graphs to enhance reasoning capability. However, existing approaches typically depend on manually designed guidance and interact with knowledge graphs through a limited set of predefined tools, which substantially constrains graph exploration. To address these limitations, we propose GraphScout, a training-centric agentic graph reasoning framework equipped with more flexible graph exploration tools. GraphScout enables models to autonomously interact with knowledge graphs to synthesize structured training data which are then used to post-train LLMs, thereby internalizing agentic graph reasoning ability without laborious manual annotation or task curation. Extensive experiments across five knowledge-graph domains show that a small model (e.g., Qwen3-4B) augmented with GraphScout outperforms baseline methods built on leading LLMs (e.g., Qwen-Max) by an average of 16.7% while requiring significantly fewer inference tokens. Moreover, GraphScout exhibits robust cross-domain transfer performance. Our code will be made publicly available [1].

CCS Concepts

• **Information systems** → **Data mining**; • **Computing methodologies** → **Knowledge representation and reasoning**.

Keywords

Large Language Models, Graph Reasoning, Post-training

ACM Reference Format:

Anonymous Author(s). 2018. GraphScout: Empowering Large Language Models with Intrinsic Exploration Ability for Agentic Graph Reasoning. In *Proceedings of Make sure to enter the correct conference title from your rights confirmation email (Conference acronym 'XX).* ACM, New York, NY, USA, 19 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Large language models (LLMs) [1, 15, 44, 50] have achieved remarkable success in question answering and complex reasoning, but they remain prone to hallucinations and lack reliable access to domain-specific or up-to-date knowledge. Retrieval-augmented generation (RAG) [39] addresses these limitations by grounding model outputs in external knowledge at inference time. While effective for unstructured text, conventional RAG pipelines struggle with structured and relational data, such as knowledge graphs, where multi-hop dependencies and structural constraints are essential. This has motivated a growing interest in graph-based retrieval and reasoning (GraphRAG) [11], which seeks to exploit graph structure for relational inference and information aggregation.

The first class of GraphRAG methods consists of passive retrieval-driven approaches. These methods [22, 51] are typically built upon the standard RAG framework. They first employ static retrieval strategies to select a set of nodes from the knowledge graph that are relevant to the given query, and then expand their neighboring subgraphs according to predefined rules, such as a fixed hop count. The resulting structured information is subsequently linearized into textual form and provided to LLMs for downstream reasoning. The second class comprises active traversal-based approaches. These methods [23, 40] equip LLMs with a set of basic graph interaction tools, such as node querying and relation expansion, and leverage carefully designed prompting schemes to enable multi-round interactions with the knowledge graph during the reasoning process. Active traversal-based approaches enable dynamic graph exploration, enhancing retrieval–reasoning coordination. Recent methods [13, 26, 32, 33] further structure this process through explicit traversal planning, schema-aware validation, or multi-agent coordination, improving robustness and accuracy for complex graph reasoning.

Despite recent progress in GraphRAG, existing methods still suffer from notable limitations in flexibility and graph exploration. On the one hand, most current approaches rely on manually designed and limited sets of graph interaction primitives, which constrains their expressiveness and flexibility. For example, when performing multi-hop neighbor queries along known path types, models often have to invoke basic neighbor expansion operations iteratively, resulting in prolonged interaction sequences and low efficiency. On the other hand, LLMs lack intrinsic priors for structured graph exploration, causing these methods to primarily depend on external mechanisms, such as workflow constraints, to enhance graph exploration. As a result, they struggle to achieve a fundamental improvement in the LLM's intrinsic graph reasoning and exploration capabilities. Empirical results reported in prior work (e.g., GraphCoT [23] in Figure 1) also reflect these limitations to some extent. Even with more flexible graph exploration tools, these issues remain unresolved. For example, PolyG [33] adopts a Cypher-based interface to enable more general graph operations, yet its performance (in Figure 1) only marginally surpasses that of GraphCoT due to its limited intrinsic graph reasoning and exploration abilities.

[1] https://anonymous.4open.science/r/GraphScout-KDD

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference acronym 'XX, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

These observations suggest that current GraphRAG research calls for a shift in methodological perspective. Rather than continually stacking increasingly complex prompt engineering or workflow control mechanisms, a more promising direction is to equip LLMs with more flexible and expressive graph interaction tools, while fundamentally enhancing their intrinsic graph reasoning and exploration abilities. Nevertheless, enhancing such intrinsic capabilities typically requires targeted post-training, which in turn relies on diverse, high-quality annotated graph reasoning trajectories that are expensive to manually construct and difficult to scale in practice, forming a major bottleneck.

To address the above challenges, we propose **GraphScout**, a training-centric framework designed to equip LLM with intrinsic exploration ability for agentic graph reasoning. GraphScout consists of three core components. First, GraphScout provides a set of **Agentic Graph Exploration Tools**, including a Code Interpreter and a Node Retriever, which enable LLMs to interact with the knowledge graph in a highly flexible and programmable manner. Second, the **Graph Quizzer** module leverages a strong LLM as a senior scout to freely explore the graph environment, automatically synthesizing diverse and high-quality graph query–answer pairs, along with their corresponding evidence clues that serve as supervision signals for subsequent training. Finally, the **Graph Solver** module optimizes small-parameter LLM as a junior scout under a multi-turn interaction paradigm, enabling it to progressively acquire agentic graph reasoning ability. Experiments show that LLMs trained with GraphScout consistently outperform baselines built on substantially larger backbone LLMs on multiple GraphRAG benchmarks by an average margin of **16.7%**, while requiring an order of magnitude fewer inference tokens. Moreover, models trained on a single-domain knowledge graph transfer robustly to unseen domains, demonstrating stable cross-domain performance.

In summary, our contributions are as follows:

- We identify the flexibility and graph exploration challenges of existing GraphRAG methods, which largely rely on manually designed interaction tools and external control mechanisms, and highlight the need for enabling LLMs with intrinsic exploration ability for agentic graph reasoning.
- We propose **GraphScout**, a training-centric framework that equips LLMs with Agentic Graph Exploration Tools for intrinsic graph reasoning. GraphScout integrates Graph Quizzer to autonomously construct diverse, high-quality graph question–answer pairs, and Graph Solver to post-train LLMs to internalize effective graph exploration ability.
- Extensive experiments across multiple GraphRAG datasets demonstrate that GraphScout enables small-parameter LLMs to achieve strong performance and efficiency, surpassing baseline methods built on flagship LLMs by an average margin of 16.7%, while exhibiting robust cross-domain generalization and reduced inference cost.

2 Related Work

2.1 LLM for Graph Reasoning

Inspired by the recent success of large language models (LLMs) in natural language processing, increasing attention has been paid to applying LLMs to graph-related tasks [22]. Existing studies can be broadly categorized according to the role LLMs play in graph reasoning. One line of work treats LLMs as feature extractors [6, 7, 10, 29, 52], where textual information associated with graph nodes or edges is encoded by LLMs and then fed into graph neural networks (GNNs) [27, 45, 49]. These methods typically adopt an LLM-GNN cascaded architecture, leveraging LLMs for semantic representation and GNNs for structural modeling. Another line of research employs LLMs as final predictors [24]. In this paradigm, graph structures are either serialized into textual sequences and processed directly by LLMs [4, 47], or combined with GNNs that are specifically trained or fine-tuned to provide structural supervision for LLM-based reasoning [2, 5, 41]. Such approaches have been applied to traditional graph tasks such as node classification and link prediction [30, 41], as well as graph algorithmic reasoning that requires structural understanding [8, 12, 17, 21, 35]. In contrast to the above approaches, our work focuses on enabling LLMs to perform explicit reasoning over knowledge graphs, rather than using graphs solely as feature sources or implicit structural cues.

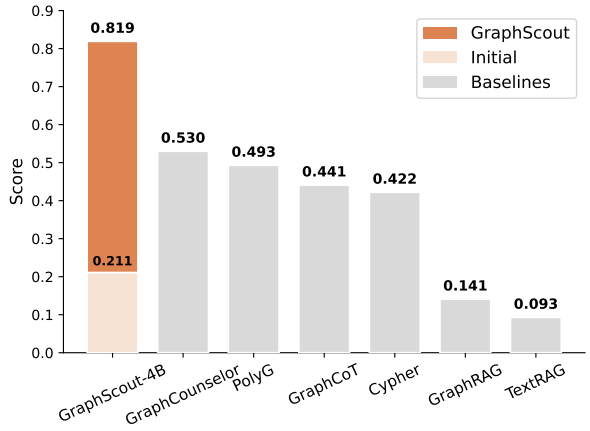

Figure 1: Qwen3-4B-Instruct with GraphScout achieves substantial gains over leading LLMs (e.g.,Qwen-Max) with prompting-based GraphRAG baselines in Healthcare dataset.

2.2 Augmenting LLMs with Knowledge Graph

Despite their strong reasoning capabilities [1, 44, 50], LLMs suffer from limitations such as hallucinations [36, 43] and outdated knowledge, motivating retrieval-augmented generation (RAG) [3, 14, 25]. While effective for unstructured text, conventional RAG struggles with relational knowledge [28, 37]. To address this gap, recent work augments LLMs with knowledge graphs, giving rise to GraphRAG [11]. Existing approaches can be broadly categorized into two categories. The first follows a passive retrieval-driven paradigm, retrieving relevant nodes or subgraphs and linearizing them into textual representations for LLM consumption [4, 11, 18]. The second line of work adopts an active traversal-based paradigm, enabling LLMs to iteratively interact with knowledge graphs by selecting traversal operations step by step [23, 40]. Representative methods include GraphCoT [23], which aligns chain-of-thought reasoning with stepwise graph traversal. More recent approaches further extend this paradigm through structured planning or agent

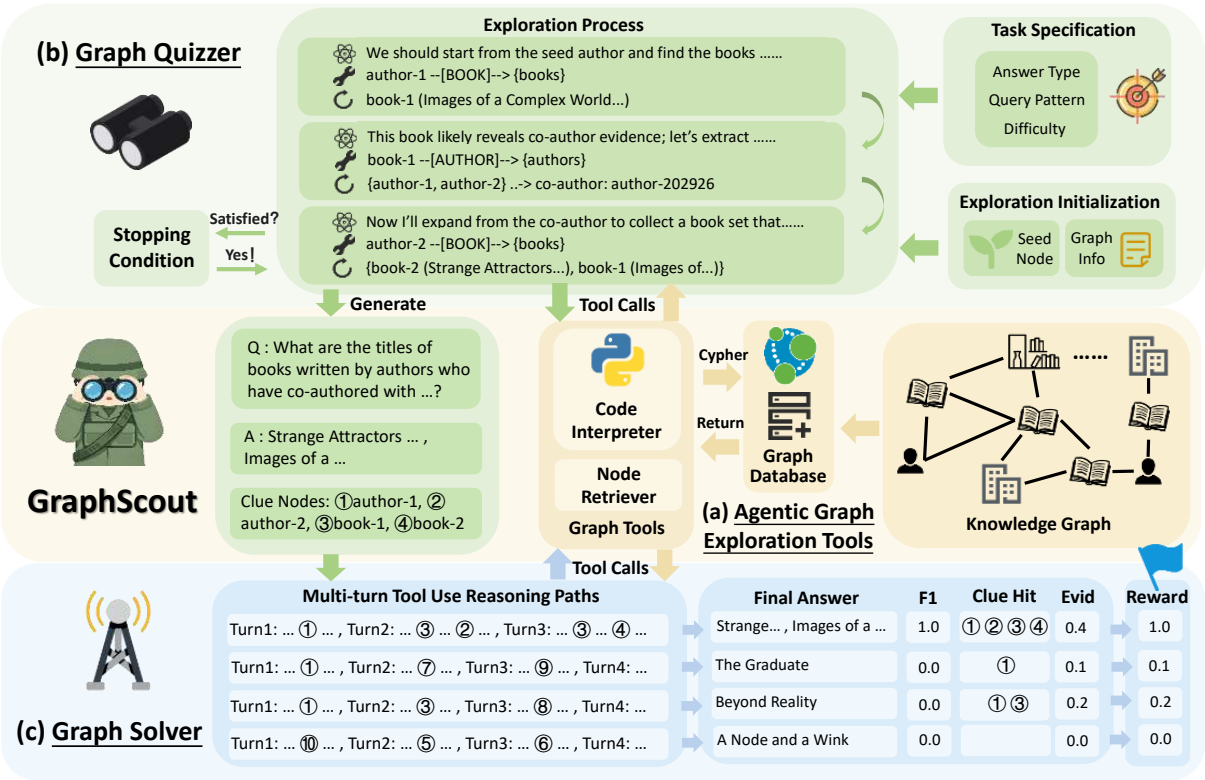

Figure 2: Overview of GraphScout framework.

collaboration [13, 26, 32, 33], improving robustness and accuracy in complex graph reasoning. PolyG [33] generates Cypher queries conditioned on the question structure, while GraphCounselor [13] leverages multi-agent coordination for planning and verification. Despite these advances, such methods still rely on manually designed interaction schemes and external control mechanisms, rather than learning intrinsic graph reasoning and exploration capabilities.

3 GraphScout

As illustrated in Figure 2, GraphScout overcomes the limitations of predefined tools by integrating a code interpreter and a node retriever as Agentic Graph Exploration Tools for both Graph Quizzer and Graph Solver. Graph Quizzer is designed to explore high-quality graph question-answer pairs for LLM post-training, employing a strong LLM as a senior scout to construct diverse graph problems along with their corresponding traversal clues. Graph Solver is proposed to endow LLMs with intrinsic exploration by post-training a small-parameter LLM as a junior scout, formulating graph reasoning as learning a multi-turn decision policy for agentic graph interaction. Training is primarily driven by answer rewards, with auxiliary clue-based rewards to guide exploration under sparse feedback. Consequently, the overall GraphScout framework comprehensively explores diverse graph reasoning problems through the Graph Quizzer and efficiently addresses intrinsic graph exploration challenges via the Graph Solver.

3.1 Preliminary

Formally, a knowledge graph is denoted by $\mathcal{K} \subseteq \mathcal{E} \times \mathcal{R} \times \mathcal{E}$, where $\mathcal{E}$ is the set of entities and $\mathcal{R}$ is the set of relations. A fact in a knowledge graph is commonly represented as a triple (h, r, t) for convenience, where $h, t \in \mathcal{E}$. The graph-based question answering task under a retrieval-augmented setting is defined as

$$f : \mathcal{Q} \times \mathcal{K} \rightarrow \mathcal{Y}, \tag{1}$$

where $\mathcal{Q}$ is the set of graph reasoning questions in natural language [23] and $\mathcal{Y}$ is the set of answers. The graph reasoning process of an LLM can be modeled as a policy $\pi(a_t \mid q, c_t)$, which selects the next action a_t given the input question q and the historical reasoning trajectory c_t. Here, $\pi(\cdot)$ is the policy distribution of an LLM, $q \in \mathcal{Q}$ is a question, $a_t \in \mathcal{A}$ is an action selection at step t, $c_t = (o_1, a_1, \cdots, o_{t-1}, a_{t-1}, o_t)$ is the contextual observation-action sequence. When the process reaches an answer, the reasoning trajectory can be denoted as $\tau = (o_1, a_1, \cdots, a_T, o_T)$, where T is the number of reasoning steps and $o_1 = q$. Existing prompting-based methods [13, 26, 32, 33] rely on external guidance and primarily focus on enriching the action space $\mathcal{A}$ with more hand-crafted tools and auxiliary prompts. However, a predefined set of tools $\mathcal{A}$ and a fixed policy $\pi(\cdot)$ significantly limit the exploration potential of LLMs. Therefore, our goal is to unlock the intrinsic exploration capability of LLMs by incorporating flexible exploration tools $\mathcal{A}_{\text{graph}}$ and learning an optimized policy $\pi_\theta(\cdot)$ for graph reasoning.

3.2 Agentic Graph Exploration Tools

As the adage states, "If a craftsman wants to do his work well, he must first sharpen his tools". Existing GraphRAG methods typically rely on a predefined set of tools, such as retrieving neighboring nodes, checking relations, or extracting node attributes. While simple, these approaches struggle with neighborhood explosion and multi-hop collaborative patterns. In contrast, GraphScout exposes an agentic graph exploration interface $\mathcal{A}_{\text{graph}}$, which integrates two complementary tools: a **Code Interpreter** for high-order pattern analysis and a **Node Retriever** for global graph retrieval. This agentic interface is invoked proactively by the LLM, formatted as

```
<tool_call> {
    "name":"tool_name",
    "arguments":{"tool_arguments"}}
</tool_call>.
```

3.2.1 ***Code Interpreter.*** The Code Interpreter provides a safe Python execution environment with access to a Cypher-based graph query interface, allowing the model to write executable code for precise and compositional graph queries. Through this interface, the model can flexibly control query logic, process intermediate results, and adapt its exploration strategy based on the evolving interaction context, rather than following predefined traversal templates. In our implementation, large-scale knowledge graphs are stored in a Neo4j database. During a preprocessing stage, each dataset is imported into Neo4j and equipped with appropriate schema-level and property-level indexes, enabling efficient execution of Cypher queries.

3.2.2 ***Node Retriever.*** The Node Retriever is implemented as a FAISS-based vector search module that supports fuzzy entity grounding. Given textual mentions in questions or intermediate hypotheses, it retrieves candidate node identifiers based on semantic similarity, enabling robust mapping from language to graph entities prior to structured querying.

By combining a programmable Code Interpreter with a lightweight Node Retriever, GraphScout extends the action space as $\mathcal{A} = \mathcal{A}_{\text{graph}} \cup \mathcal{L}$, where $\mathcal{L}$ is the intrinsic verbal reasoning actions of an LLM. Therefore, GraphScout significantly expands the exploration space available to the LLM while keeping the tool interface minimal. More details are provided in Appendix B.4.1.

3.3 Graph Quizzer

Using these agentic graph exploration tools $\mathcal{A}_{\text{graph}}$, Graph Quizzer employs a strong LLM as a *senior scout* to flexibly explore graph problems $\mathcal{Q}_{\text{train}}$ at varying difficulty levels according to general task specifications. As shown in the upper panel of Figure 2, Graph Quizzer explores the higher-order neighborhoods and global graph semantics by alternating verbal reasoning and tool use until a stopping condition is met. Through this exploration loop, Graph Quizzer generates high-quality graph question-answering pairs $\mathcal{D}_{\text{train}} = (\mathcal{Q}_{\text{train}}, \mathcal{Y}_{\text{train}}, \mathcal{C}_{\text{train}})$ for subsequent LLM post-training, effectively eliminating laborious dataset curation. We explicitly record intermediate node clues $\mathcal{C}_{\text{train}} = \{c | c = (e_i, e_j, \cdots, e_k)$ and $e_i \in \mathcal{E}\}$ to mitigate reward sparsity during the ensuing training process.

3.3.1 ***Task specification.*** Each quizzer episode begins by sampling an *exploration objective* that specifies the structural and semantic requirements of the graph question to be generated. Rather than relying on using fixed templates, objectives are specified at an abstract level to guide task-oriented exploration while preserving flexibility. Each objective is characterized by a combination of **answer type**, **query pattern**, and **difficulty**, which together ensure diversity among generated questions.

Answer type. The answer type defines the expected form of the final answer and the corresponding supervision signal. We consider four common types: `entity` (identifying a graph entity), `boolean` (judging whether a given statement is true or false), `number` (counting or aggregation), and `set` (a collection of entities). This diversity encourages the quizzer to produce supervision beyond simple entity retrieval.

Query pattern. The query pattern specifies the abstract structural form of a graph query, indicating which components are given and which must be discovered through exploration. Following PolyG [33], we adopt standard patterns including $\langle h, _, _ \rangle$, $\langle h, r, * \rangle$, $\langle h, *, t \rangle$, and $\langle h, r, t \rangle$, as well as hybrid patterns that combine multiple forms to support more compositional reasoning.

Difficulty. The difficulty level controls the required reasoning depth and is categorized as `simple`, `medium`, or `hard`, corresponding to single-hop to deeper multi-hop reasoning. Varying difficulty allows quizzer to generate exploration trajectories with different lengths and structural complexity.

Together, these three factors define a structured yet flexible objective space that enables Graph Quizzer to cover diverse graph reasoning behaviors without relying on rigid, hand-crafted task templates. Additional details are provided in Appendix B.3.

3.3.2 ***Exploration initialization.*** Given a sampled exploration objective, Graph Quizzer initializes the scout with a compact *environment context* c_{env} and a concrete *seed node* e_i, providing the minimal context required for task-oriented exploration.

Graph description. The scout is provided with a concise description of the target knowledge graph, including core schema-level information such as node types, edge types, and commonly used properties. This description also specifies the underlying exploration tools $\mathcal{A}_{\text{graph}}$, grounding the scout's subsequent interactions in a concrete execution environment.

Seed node. Each exploration episode is anchored at a *seed node* $e_i \in \mathcal{E}$ sampled from the target graph. To mitigate bias from imbalanced node-type distributions, we adopt a two-stage sampling strategy: first uniformly sampling a node type, followed by random selection of a node instance within that type. This strategy ensuring balanced coverage across graph structures and reasoning scenarios.

3.3.3 ***Exploration process.*** Starting from the initialized seed node e_i, the scout performs a bounded, multi-step exploration process conditioned on the sampled exploration objective. Exploration proceeds as an iterative *explore-assess* loop, in which the scout actively retrieves graph evidence and continuously evaluates its progress toward satisfying the objective.

At each step, the scout formulates a concrete retrieval action, such as inspecting neighboring relations, expanding along selected paths, or aggregating graph statistics, and executes it through the graph interaction interface. The returned results are summarized

into a local exploration state, which is used to assess whether the accumulated evidence is sufficient to meet the specified answer type, query pattern, and difficulty constraints.

This process is explicitly task-oriented: rather than exhaustively traversing the graph $\mathcal{K}$, the senior scout prioritizes a partial reasoning trajectory c_t that is most relevant to the current objective and terminates exploration early once sufficient supporting information has been collected. To prevent unbounded interaction, the exploration length is capped by a fixed budget of tool calls (we use T=10), encouraging efficient and focused exploration behaviors.

3.3.4 ***Question reporting.*** Once the senior scout determines that the exploration objective has been satisfied, it transitions from the question-exploration stage to the question-reporting stage. Using the collected reasoning trajectory c_T, the senior scout constructs a natural-language graph question q together with its corresponding answer y, ensuring that y is directly verifiable on the underlying graph. In addition to the final question-answer pair (q, y), Graph Quizzer records a set of *evidence clues* $c = (e_i, e_j, \cdots, e_k)$, each element being a node identifier explicitly used during question and answer formation. These clues provide an explicit link between the generated answer y and concrete graph entities e_i encountered during exploration. As a result, each quizzer episode generates a structured supervision tuple: (`question`, `answer`, `clue_nodes`), where the question q is expressed in natural language, the answer y is grounded in the graph, and the evidence clues c anchor supervision to explicit graph-level facts. This design enables downstream models to learn not only to produce correct answers, but also to align their reasoning processes with relevant evidence clues.

3.4 Graph Solver

Graph Solver further trains a small-parameter LLM as a *junior scout* to tackle the graph-reasoning questions Q_{train} proposed by Graph Quizzer, using reward signals for answer correctness and alignment with evidence clues. As shown in the bottom panel of Figure 2, the junior scout iteratively reasons the possible solution paths via exploration tools $\mathcal{A}_{\text{graph}}$ in a multi-turn paradigm until it outputs a final answer. The reasoning policy can be denoted by $\pi_\theta(\tau|q)$, where $\pi_\theta(\cdot)$ is the parameterized LLM and τ is the trajectory involving verbal reasoning steps and tool actions. This simple yet effective design empowers LLMs to perform intrinsic exploration over questions Q_{train} and complex graphs $\mathcal{K}$, going beyond the limited tools and static prompts used in existing GraphRAG methods.

3.4.1 ***Problem formulation.*** Graph Solver formulates graph question answering as learning a *multi-turn decision policy* for task-oriented interaction with a knowledge graph. Given a natural language question q, the model must decide how to sequentially invoke tools $\mathcal{A}_{\text{graph}}$ to retrieve evidence, aggregate information, and ultimately produce an answer. Formally, each episode corresponds to a trajectory $\tau = (o_1, a_1, o_2, a_2, \ldots, o_T, a_T)$ of alternating observations and actions, where the initial observation is the question itself, i.e., $o_1 = q$. Actions $\{a_t\}_{t=1}^{T}$ are expressed as tool calls, and observations $\{o_t\}_{t=1}^{T}$ include structured tool responses and the model's intermediate verbal reasoning. The objective is not to follow predefined traversal rules, but to acquire a generalizable solver strategy $\pi_\theta(\tau \mid q)$ that coordinates entity grounding, graph traversal, and answer synthesis under a fixed interaction budget.

3.4.2 ***Reward design.*** Graph Solver is trained with a trajectory-level reward that combines (i) answer correctness and (ii) evidence clues alignment, with strict format gating. First, we extract the output answer $\hat{y}$ from the final trajectory τ, and then retrieve the corresponding node set of evidence clues $c \in C_{\text{train}}$ (i.e., `clue_nodes`) collected by Graph Quizzer for reward computation.

Answer reward. We first extract the predicted answer $\hat{y}$ from a required wrapper `\answer{...}`. If the wrapper is missing, the reward is set to zero:

$$\hat{y} = \text{ExtractAnswer}(\tau), \qquad \hat{y} = \emptyset \;\Rightarrow\; r(\tau) = 0. \tag{2}$$

For valid outputs from the trajectory τ, we compute an answer reward using an F1-based metric:

$$r_{\text{ans}}(\tau) = \text{F1}(\hat{y}, y) \in [0, 1]. \tag{3}$$

The answer reward directly measures answer-level correctness and serves as the primary training signal, ensuring that the optimization objective remains aligned with the final task performance, where the F1 score measures token-level overlap.

Evidence clues-based reward. To guide exploration behavior, we introduce an evidence clues-based process reward. Let $\mathbb{I}(\tau, e_i)$ indicate whether the model's trajectory τ references or interacts with evidence node e_i. The evidence reward is then defined as:

$$r_{\text{clue}}(\tau) = \frac{1}{|c|} \sum_{i=1}^{|c|} \mathbb{I}(\tau, e_i). \tag{4}$$

Relying solely on answer-based rewards yields sparse supervision signals for multi-turn graph reasoning. An auxiliary evidence-based reward provides denser and graph-grounded intermediate supervision that guides the model's intermediate steps.

The final reward follows a case-based rule that applies the evidence reward *only* when outcome quality is low and caps the combined score in this regime:

$$r(\tau) = \begin{cases} 0, & \hat{y} = \emptyset, \\ r_{\text{ans}}(\tau), & r_{\text{ans}}(\tau) \geq \delta, \\ \min\big(r_{\text{ans}}(\tau) + r_{\text{clue}}(\tau),\ \delta\big), & r_{\text{ans}}(\tau) < \delta, \end{cases} \tag{5}$$

where δ is a threshold used to identify low-accuracy predictions and also serves as the upper bound for the bonus-augmented regime. This design ensures that (i) malformed outputs receive no reward, (ii) answer correctness remains the primary training signal, and (iii) evidence alignment provides auxiliary guidance without allowing low-accuracy trajectories to obtain high rewards.

3.4.3 ***Optimization.*** Graph Solver is trained using multi-turn reinforcement learning, where the model performs multiple reasoning and tool-use steps before producing a final answer. We employ Group Relative Policy Optimization (GRPO) [16] to optimize the policy. For each question q, we sample a group of tool-mediated interaction trajectories $G = \{\tau_1, \ldots, \tau_{|G|}\}$ from the current policy π_θ. Each trajectory τ_i receives a scalar reward r_i computed as described in Section 3.4.2, while intermediate tool execution results do not directly contribute to the loss. For each trajectory τ_i, we compute a normalized relative advantage $A_i = (r_i - \text{mean}\{r_k\}_{k=1}^{|G|})/\text{std}\{r_k\}_{k=1}^{|G|}$.

Table 1: Performance Comparison on GRBENCH Across Baseline Methods and GraphScout. Relative improvement is measured against the strongest baseline on each dataset. On QwenScore, GraphScout achieves a 16.7% average gain.

Method	Dataset	Healthcare		Literature		Academic		E-Commerce		Legal	
	LLM	QwenScore	F1	QwenScore	F1	QwenScore	F1	QwenScore	F1	QwenScore	F1
BaseLLM	GPT-4o	0.137	0.048	0.221	0.064	0.097	0.080	0.110	0.080	0.244	0.110
	GLM-4.6	0.107	0.099	0.179	0.102	0.095	0.110	0.060	0.111	0.222	0.243
	Qwen-Max	0.074	0.065	0.158	0.098	0.112	0.128	0.060	0.105	0.244	0.298
	DeepSeek-Chat	0.104	0.075	0.192	0.111	0.134	0.140	0.095	0.115	0.222	0.265
TextRAG	GPT-4o	0.074	0.059	0.179	0.116	0.098	0.090	0.200	0.181	0.256	0.232
	GLM-4.6	0.093	0.085	0.146	0.091	0.092	0.105	0.250	0.241	0.217	0.253
	Qwen-Max	0.067	0.085	0.142	0.092	0.092	0.106	0.090	0.148	0.344	0.406
	DeepSeek-Chat	0.085	0.060	0.196	0.137	0.146	0.158	0.115	0.138	0.333	0.365
GraphRAG	GPT-4o	0.156	0.129	0.217	0.136	0.105	0.092	0.315	0.308	0.239	0.233
	GLM-4.6	0.156	0.174	0.213	0.164	0.124	0.135	0.325	0.350	0.172	0.253
	Qwen-Max	0.141	0.168	0.208	0.166	0.114	0.130	0.180	0.243	0.356	0.424
	DeepSeek-Chat	0.170	0.140	0.221	0.151	0.177	0.164	0.165	0.167	0.361	0.381
Cypher	GPT-4o	0.311	0.107	0.533	0.157	0.605	0.197	0.485	0.101	0.389	0.118
	GLM-4.6	0.319	0.104	0.583	0.145	0.538	0.147	0.490	0.114	0.422	0.106
	Qwen-Max	0.407	0.131	0.558	0.143	0.572	0.181	0.520	0.117	0.406	0.132
	DeepSeek-Chat	0.422	0.097	0.583	0.144	0.580	0.186	0.545	0.114	0.350	0.094
GraphCoT	GPT-4o	0.415	0.447	0.463	0.395	0.587	0.547	0.410	0.360	0.494	0.403
	GLM-4.6	0.396	0.475	0.504	0.486	0.629	0.614	0.380	0.388	0.511	0.517
	Qwen-Max	0.415	0.470	0.479	0.437	0.608	0.570	0.465	0.418	0.533	0.531
	DeepSeek-Chat	0.441	0.443	0.563	0.466	0.628	0.546	0.470	0.418	0.555	0.559
PolyG	GPT-4o	0.378	0.107	0.463	0.119	0.516	0.138	0.440	0.092	0.344	0.087
	GLM-4.6	0.467	0.101	0.504	0.090	0.593	0.129	0.445	0.090	0.356	0.079
	Qwen-Max	0.341	0.104	0.471	0.100	0.499	0.153	0.430	0.098	0.317	0.074
	DeepSeek-Chat	0.493	0.104	0.533	0.110	0.585	0.146	0.445	0.087	0.339	0.070
GraphCounselor	GPT-4o	0.485	0.441	0.592	0.438	0.644	0.605	0.490	0.381	0.433	0.219
	GLM-4.6	0.407	0.421	0.579	0.457	0.595	0.589	0.490	0.459	0.339	0.246
	Qwen-Max	0.530	0.484	0.625	0.524	0.637	0.639	0.459	0.447	0.478	0.262
	DeepSeek-Chat	0.470	0.452	0.621	0.578	0.657	0.659	0.510	0.489	0.483	0.286
GraphScout	4B w/o train	0.211	0.217	0.179	0.148	0.210	0.220	0.250	0.255	0.189	0.226
	4B w/ train	**0.819**	**0.855**	**0.696**	**0.646**	**0.710**	**0.739**	0.575	0.562	**0.567**	**0.597**
	8B w/o train	0.403	0.415	0.396	0.386	0.422	0.401	0.365	0.358	0.361	0.364
	8B w/ train	0.770	0.783	0.663	0.628	0.685	0.702	**0.585**	**0.569**	0.555	0.573
Relative Improvement		**54.6%**	**76.5%**	**11.4%**	**11.8%**	**8.2%**	**12.0%**	**7.3%**	**16.3%**	**2.2%**	**6.8%**

The policy parameters θ are then updated by maximizing a clipped surrogate objective with a KL penalty:

$$L(\theta) = \frac{1}{|G|} \sum_{i=1}^{|G|} \Big[\min \big(\rho_i A_i,\ \mathrm{clip}(\rho_i, 1-\varepsilon, 1+\varepsilon) A_i\big) - \beta\, D_{\mathrm{KL}}(\pi_\theta \,\|\, \pi_{\mathrm{ref}}) \Big], \tag{6}$$

where $\rho_i = \pi_\theta(\tau_i \mid q)/\pi_{\theta_{\mathrm{old}}}(\tau_i \mid q)$ is the importance ratio computed from model-generated tokens, ε controls the clipping range, and β weights the KL regularization term that constrains deviation from the reference policy π_{ref}. More details are provided in Appendix A.5.

4 Experiment

4.1 Experimental Setup

Dataset. We conduct experiments on GRBENCH dataset [23] following baseline approaches [13, 23], which contains graphs from five domains and a total of 1,740 questions categorized into three difficulty levels: easy, medium, and hard. All questions are in English. More details can be found in the Appendix A.3.

Baselines. We compare our GraphScout approach with several baselines: BaseLLM, TextRAG [14], GraphRAG [51], Cypher [33], GraphCoT [23], PolyG [33], and GraphCounselor [13]. To ensure a comprehensive and fair evaluation of these baselines, we instantiate each method with multiple state-of-the-art LLMs, including GPT-4o [1], GLM-4.6 [15], Qwen-Max [50], and DeepSeek-Chat [31]. Detailed settings are provided in the Appendix A.4.

Evaluation Metrics. We evaluate model performance using both model-based and automatic metrics. Specifically, we employ QwenScore, an LLM-based metric using Qwen-Max as a judge to assess answer correctness by comparing generated outputs with ground-truth answers, and report the proportion of responses judged as correct. Besides, we report the F1 score, which measures token-level overlap between predicted and ground-truth answers.

Models and Training Settings. We conduct experiments using two model variants: Qwen3-4B-Instruct-2507 and Qwen3-8B. For the Graph Quizzer stage, we use DeepSeek-Chat to explore the knowledge graph and synthesize sufficient training data. To prevent test-set leakage, we ensure that all clue nodes in the training set are disjoint from any nodes appearing in the test set questions.

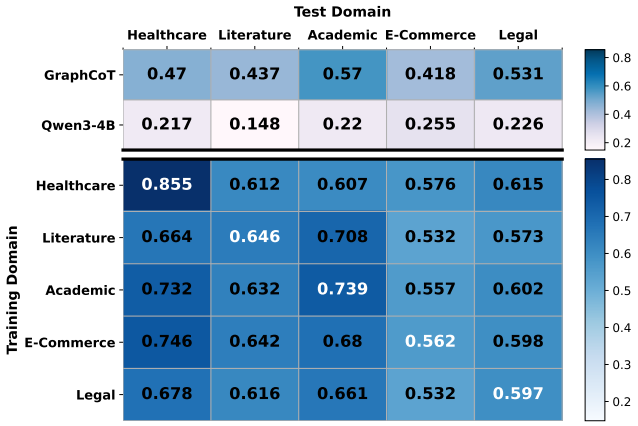

Figure 3: Cross-Domain Generalization Performance Across Training and Test Domains.

During training, we adopt the `verl` [38] framework and employ its implementation of Group Relative Policy Optimization (GRPO) [16] for reinforcement learning, training the models for 400 optimization steps. Detailed training hyperparameters and configurations are provided in the Appendix A.1.

4.2 Performance Analysis

4.2.1 ***Overall Accuracy Analysis.*** Table 1 reports performance on GRBENCH across five domains using both QwenScore and F1. Overall, graph-based approaches consistently outperform BaseLLM and TextRAG, underscoring the importance of explicitly leveraging structured graph knowledge. Methods such as Cypher and PolyG achieve relatively high QwenScore but noticeably lower F1, which is expected as they do not enforce strict answer format constraints. In such cases, QwenScore, which emphasizes semantic equivalence, better reflects answer quality.

Despite relying on substantially smaller backbone LLMs, GraphScout achieves the strongest overall performance after post-training, surpassing all baseline methods built upon leading LLMs such as DeepSeek-Chat and Qwen-Max. This advantage is particularly pronounced on the Healthcare dataset, which features a highly complex graph structure with 11 node types and 24 edge types. Under such conditions, baseline methods, even when supported by powerful LLM backbones, are constrained by limited graph exploration capabilities and thus struggle to reason effectively over intricate relational structures. In contrast, GraphScout is explicitly designed to support agentic graph reasoning by equipping models with intrinsic exploration ability and flexible graph interaction tools, making it especially well suited to reasoning over complex graphs. As a result, it demonstrates particularly strong performance in this dataset. Case studies are provided in Appendix B.2.

Regarding model scale, GraphScout-8B outperforms GraphScout-4B without training. After post-training, however, the 4B variant achieves stronger performance than the 8B model in most cases, consistent with observations from prior post-training studies [9]. We hypothesize that this behavior may be related to the hybrid reasoning configuration of Qwen3-8B.

Table 2: Ablation study of GraphScout ("rw" denotes random walk on knowledge graphs).

	Healthcare		Literature	
	QwenScore	F1	QwenScore	F1
GraphScout	**0.819**	**0.855**	**0.696**	**0.646**
w/o Graph Solver	0.211	0.217	0.210	0.220
w/o $\mathcal{A}_{\text{code}}$	0.107	0.101	0.196	0.148
w/o r_{clue}	0.785	0.812	0.667	0.635
rw Graph Quizzer	0.678	0.705	0.608	0.571

4.2.2 ***Cross-Domain Generalization Analysis.*** We further evaluate the robustness of GraphScout under cross-domain settings to examine whether the learned exploration capability transfers beyond the training domain. Figure 3 reports the F1-score performance of GraphScout-4B when trained on a single domain and evaluated on all five GRBENCH domains. For comparison, we also include the performance of Qwen-Max based GraphCoT and the base Qwen3-4B-Instruct evaluated independently on each domain.

Overall, GraphScout exhibits robust cross-domain performance across all training–testing combinations, substantially outperforming the baselines in most cases. Notably, models trained on one domain consistently achieve solid performance on unseen domains, with only mild performance degradation under domain shifts. This indicates that GraphScout learns transferable, intrinsic graph exploration behaviors rather than memorizing domain-specific structures or traversal heuristics, enabling effective adaptation across heterogeneous graphs without additional efforts.

4.2.3 ***Ablation Analysis.*** Table 2 reports an ablation study of GraphScout. Removing Graph Solver (*w/o Graph Solver*) leads to substantial performance drops, confirming that the gains stem from learned exploration behaviors rather than pretrained knowledge alone. Eliminating Code Interpreter (*w/o* $\mathcal{A}_{\text{code}}$) causes the most severe degradation, highlighting the necessity of tool-mediated graph interaction. Removing Evidence reward (*w/o* r_{clue}) consistently reduces performance, indicating that graph evidence provides important process-level guidance beyond final answer correctness. Finally, replacing Graph Quizzer with random-walk-based question generation [34] (*rw Graph Quizzer*) also degrades performance. Here, training data are generated by sampling subgraphs through random walks, converting them into text, and prompting an LLM to produce question–answer pairs without interactive exploration. The performance drop suggests that supervision derived from static subgraph sampling provides weaker guidance for learning effective exploration behaviors.

4.2.4 ***Accuracy Across Question Difficulty Levels.*** To better understand how GraphScout behaves under different reasoning requirements, we further break down the GRBENCH results by difficulty level (Figure 4). In GRBENCH, **Easy** questions mainly test basic fact retrieval and single-step reasoning; **Medium** questions require multi-step traversal, more complex path discovery, and evidence aggregation; **Hard** questions emphasize more inductive reasoning and are often recommendation-like.

Easy. GraphScout-4B consistently achieves strong performance across all five domains on easy questions, often outperforming

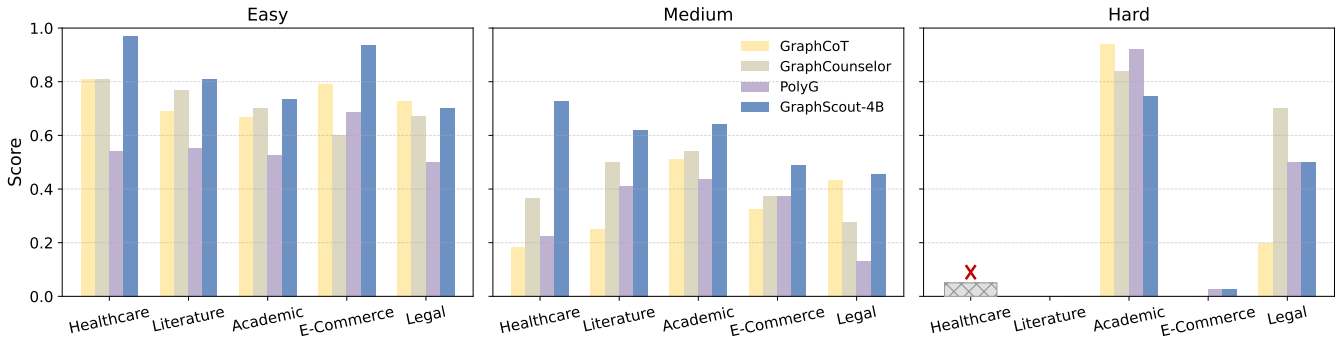

Figure 4: Performance Comparison Across Question Difficulty Levels on GRBENCH using F1-score as the metric. The Healthcare domain contains no hard questions. For the Literature, all methods achieve an F1-score of 0% on hard questions.

the strongest graph-based baselines. This suggests that the model learns to reliably ground answers on graph information for common single-hop or shallow retrieval queries.

Medium. The advantage of GraphScout becomes even more evident on medium difficulty. Across domains, GraphScout-4B outperforms the baselines by a substantial margin, indicating strong capability in multi-turn exploration and multi-hop information aggregation. These results align with our motivation that learning intrinsic exploration ability (rather than relying on prompting heuristics) is particularly beneficial when the question requires iterative tool use and compositional reasoning.

Hard. Hard questions are challenging for all methods. Many hard instances in this dataset emphasize recommendation-style reasoning and external world knowledge, placing less emphasis on structured graph traversal. Consequently, the benefits of stronger graph exploration ability are less pronounced, and GraphScout-4B shows limited gains on this split.

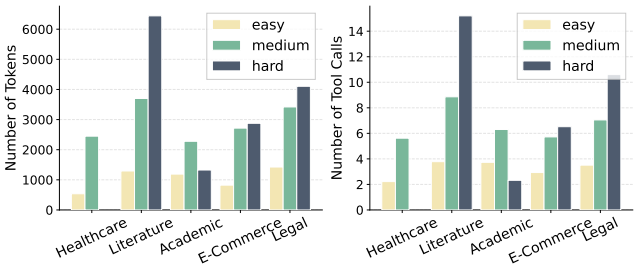

Figure 5: Behavioral analysis across question difficulty levels.

4.2.5 ***Reasoning Behavior Across Question Difficulty Levels.*** Figure 5 reports GraphScout's behavior across question difficulty levels using average output length and tool calls per question. In most domains, both metrics increase from easy to hard, suggesting deeper reasoning and more extensive graph interaction as task complexity rises. An exception occurs in the Academic domain for hard questions: these instances primarily involve recommendations within a constrained publication scope, emphasizing semantic understanding rather than explicit graph querying.

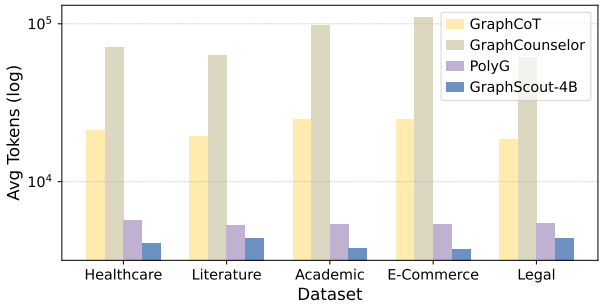

Figure 6: Average token consumption of different methods.

4.2.6 ***Efficiency Analysis.*** Figure 6 compares the average token consumption of different methods across GRBENCH domains. Overall, GraphScout-4B consistently uses substantially fewer tokens than other active traversal-based baselines. Importantly, this reduction does not come at the cost of performance; as shown in previous results, GraphScout achieves superior accuracy while maintaining significantly lower computational overhead.

5 Conclusion

This paper presents GraphScout, a framework that enhances LLMs' agentic graph reasoning with intrinsic exploration ability. GraphScout consists of three core components: Agentic Graph Exploration Tools, which equip LLMs with more flexible and programmable graph interaction interface; Graph Quizzer, which explores diverse and high-quality problem sets on graphs while recording traversal clues; and Graph Solver, which post-trains LLMs to acquire intrinsic graph exploration ability. Extensive experiments against *state-of-the-art* GraphRAG methods demonstrate that small-parameter LLMs equipped with GraphScout significantly outperforms substantially larger LLMs by an average of 16.7%. Moreover, GraphScout exhibits promising cross-domain generalization capability, indicating that LLMs can develop generalized graph reasoning ability even with training on a single graph. Future work will investigate LLMs bootstrapping via iterative self-questioning and self-answering in an iterative enhancement loop, while addressing challenges such as training instability and self-bias amplification.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[2] Ziwei Chai, Tianjie Zhang, Liang Wu, Kaiqiang Han, Xiaohai Hu, Xuanwen Huang, and Yang Yang. 2025. Graphllm: Boosting graph reasoning ability of large language model. *IEEE Transactions on Big Data* (2025).

[3] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. 2024. Benchmarking large language models in retrieval-augmented generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17754–17762.

[4] Nuo Chen, Yuhan Li, Jianheng Tang, and Jia Li. 2024. Graphwiz: An instruction-following language model for graph computational problems. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 353–364.

[5] Runjin Chen, Tong Zhao, Ajay Jaiswal, Neil Shah, and Zhangyang Wang. 2024. Llaga: Large language and graph assistant. *arXiv preprint arXiv:2402.08170* (2024).

[6] Zhikai Chen, Haitao Mao, Hang Li, Wei Jin, Hongzhi Wen, Xiaochi Wei, Shuaiqiang Wang, Dawei Yin, Wenqi Fan, Hui Liu, et al. 2024. Exploring the potential of large language models (llms) in learning on graphs. *ACM SIGKDD Explorations Newsletter* 25, 2 (2024), 42–61.

[7] Eli Chien, Wei-Cheng Chang, Cho-Jui Hsieh, Hsiang-Fu Yu, Jiong Zhang, Olgica Milenkovic, and Inderjit S Dhillon. 2021. Node feature extraction by self-supervised multi-scale neighborhood prediction. *arXiv preprint arXiv:2111.00064* (2021).

[8] Xinnan Dai, Haohao Qu, Yifen Shen, Bohang Zhang, Qihao Wen, Wenqi Fan, Dongsheng Li, Jiliang Tang, and Caihua Shan. 2024. How do large language models understand graph patterns? a benchmark for graph pattern comprehension. *arXiv preprint arXiv:2410.05298* (2024).

[9] distillabs. 2025. We benchmarked 12 small language models across 8 tasks to find the best base model for fine-tuning. https://www.distillabs.ai/blog/we-benchmarked-12-small-language-models-across-8-tasks-to-find-the-best-base-model-for-fine-tuning.

[10] Keyu Duan, Qian Liu, Tat-Seng Chua, Shuicheng Yan, Wei Tsang Ooi, Qizhe Xie, and Junxian He. 2023. Simteg: A frustratingly simple approach improves textual graph learning. *arXiv preprint arXiv:2308.02565* (2023).

[11] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. 2024. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130* (2024).

[12] Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2023. Talk like a graph: Encoding graphs for large language models. *arXiv preprint arXiv:2310.04560* (2023).

[13] Junqi Gao, Xiang Zou, Ying Ai, Dong Li, Yichen Niu, Biqing Qi, and Jianxing Liu. 2025. Graph Counselor: Adaptive Graph Exploration via Multi-Agent Synergy to Enhance LLM Reasoning. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 24650–24668. doi:10.18653/v1/2025.acl-long.1202

[14] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* 2, 1 (2023).

[15] Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793* (2024).

[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[17] Xiaojun Guo, Ang Li, Yifei Wang, Stefanie Jegelka, and Yisen Wang. 2025. G1: Teaching LLMs to Reason on Graphs with Reinforcement Learning. *arXiv preprint arXiv:2505.18499* (2025).

[18] Zirui Guo, Lianghao Xia, Yanhua Yu, Tu Ao, and Chao Huang. 2024. Lightrag: Simple and fast retrieval-augmented generation. *arXiv preprint arXiv:2410.05779* (2024).

[19] Ruining He and Julian McAuley. 2016. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*. 507–517.

[20] Daniel Scott Himmelstein, Antoine Lizee, Christine Hessler, Leo Brueggeman, Sabrina L Chen, Dexter Hadley, Ari Green, Pouya Khankhanian, and Sergio E Baranzini. 2017. Systematic integration of biomedical knowledge prioritizes drugs for repurposing. *elife* 6 (2017), e26726.

[21] Zixiao Huang, Lifeng Guo, Wenhao Li, Junjie Sheng, Chuyun Shen, Haosheng Chen, Bo Jin, Changhong Lu, and Xiangfeng Wang. 2025. GraphThought: Graph Combinatorial Optimization with Thought Generation. *arXiv preprint arXiv:2502.11607* (2025).

[22] Bowen Jin, Gang Liu, Chi Han, Meng Jiang, Heng Ji, and Jiawei Han. 2024. Large language models on graphs: A comprehensive survey. *IEEE Transactions on Knowledge and Data Engineering* (2024).

[23] Bowen Jin, Chulin Xie, Jiawei Zhang, Kashob Kumar Roy, Yu Zhang, Zheng Li, Ruirui Li, Xianfeng Tang, Suhang Wang, Yu Meng, et al. 2024. Graph chain-of-thought: Augmenting large language models by reasoning on graphs. *arXiv preprint arXiv:2404.07103* (2024).

[24] Bowen Jin, Wentao Zhang, Yu Zhang, Yu Meng, Xinyang Zhang, Qi Zhu, and Jiawei Han. 2023. Patton: Language model pretraining on text-rich networks. *arXiv preprint arXiv:2305.12268* (2023).

[25] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering.. In *EMNLP (1)*. 6769–6781.

[26] Savini Kashmira, Jayanaka L Dantanarayana, KrisztiĂąn Flautner, Lingjia Tang, and Jason Mars. 2025. GraphRunner: A Multi-Stage Framework for Efficient and Accurate Graph-Based Retrieval. *arXiv preprint arXiv:2507.08945* (2025).

[27] TN Kipf. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[28] Jérôme Kunegis. 2013. Konect: the koblenz network collection. In *Proceedings of the 22nd international conference on world wide web*. 1343–1350.

[29] Wenda Li, Tongya Zheng, Shunyu Liu, Yu Wang, Kaixuan Chen, Hanyang Yuan, Bingde Hu, Zujie Ren, Mingli Song, and Gang Chen. 2025. Towards Efficient LLM-aware Heterogeneous Graph Learning. *arXiv preprint arXiv:2511.17923* (2025).

[30] Tianqianjin Lin, Pengwei Yan, Kaisong Song, Zhuoren Jiang, Yangyang Kang, Jun Lin, Weikang Yuan, Junjie Cao, Changlong Sun, and Xiaozhong Liu. 2024. Langgfm: A large language model alone can be a powerful graph foundation model. *arXiv preprint arXiv:2410.14961* (2024).

[31] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024).

[32] Lihui Liu. 2025. Graph-O1: Monte Carlo Tree Search with Reinforcement Learning for Text-Attributed Graph Reasoning. *arXiv preprint arXiv:2512.17912* (2025).

[33] Renjie Liu, Haitian Jiang, Xiao Yan, Bo Tang, and Jinyang Li. 2025. PolyG: Effective and Efficient GraphRAG with Adaptive Graph Traversal. *arXiv preprint arXiv:2504.02112* (2025).

[34] Rui Lu, Zhenyu Hou, Zihan Wang, Hanchen Zhang, Xiao Liu, Yujiang Li, Shi Feng, Jie Tang, and Yuxiao Dong. 2025. Deepdive: Advancing deep search agents with knowledge graphs and multi-turn rl. *arXiv preprint arXiv:2509.10446* (2025).

[35] Zihan Luo, Xiran Song, Hong Huang, Jianxun Lian, Chenhao Zhang, Jinqi Jiang, Xing Xie, and Hai Jin. 2024. Graphinstruct: Empowering large language models with graph understanding and reasoning capability. *arXiv preprint arXiv:2403.04483* (2024).

[36] Vipula Rawte, Amit Sheth, and Amitava Das. 2023. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922* (2023).

[37] Benedek Rozemberczki and Rik Sarkar. 2020. Characteristic Functions on Graphs: Birds of a Feather, from Statistical Descriptors to Parametric Models. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM '20)*. ACM, 1325–1334.

[38] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. HybridFlow: A Flexible and Efficient RLHF Framework. *arXiv preprint arXiv: 2409.19256* (2024).

[39] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. 2021. Retrieval augmentation reduces hallucination in conversation. *arXiv preprint arXiv:2104.07567* (2021).

[40] Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel M Ni, Heung-Yeung Shum, and Jian Guo. 2023. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. *arXiv preprint arXiv:2307.07697* (2023).

[41] Jiabin Tang, Yuhao Yang, Wei Wei, Lei Shi, Lixin Su, Suqi Cheng, Dawei Yin, and Chao Huang. 2024. Graphgpt: Graph instruction tuning for large language models. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 491–500.

[42] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 990–998.

[43] SMTI Tonmoy, SM Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. 2024. A comprehensive survey of hallucination mitigation techniques in large language models. *arXiv preprint arXiv:2401.01313* 6 (2024).

[44] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[45] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).
[46] Mengting Wan and Julian McAuley. 2018. Item recommendation on monotonic behavior chains. In *Proceedings of the 12th ACM conference on recommender systems*. 86–94.
[47] Heng Wang, Shangbin Feng, Tianxing He, Zhaoxuan Tan, Xiaochuang Han, and Yulia Tsvetkov. 2023. Can language models solve graph problems in natural language? *Advances in Neural Information Processing Systems* 36 (2023), 30840–30861.
[48] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft academic graph: When experts are not enough. *Quantitative Science Studies* 1, 1 (2020), 396–413.
[49] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826* (2018).
[50] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388* (2025).
[51] Ruosong Ye, Caiqi Zhang, Runhui Wang, Shuyuan Xu, and Yongfeng Zhang. 2024. Language is All a Graph Needs. In *Findings of the Association for Computational Linguistics: EACL 2024*, Yvette Graham and Matthew Purver (Eds.). Association for Computational Linguistics, St. Julian's, Malta, 1955–1973. https://aclanthology.org/2024.findings-eacl.132/
[52] Jianan Zhao, Meng Qu, Chaozhuo Li, Hao Yan, Qian Liu, Rui Li, Xing Xie, and Jian Tang. 2022. Learning on large-scale text-attributed graphs via variational inference. *arXiv preprint arXiv:2210.14709* (2022).

A Implementation Details

A.1 RL Hyperparameters

The experimental settings [38] for training Base model using our GraphScout framework are shown in Table 3.

Table 3: GRPO Training Settings for GraphScout.

Method	Hyperparameters
GraphScout	RL Algorithm = GRPO n_resp_per_prompt = 8 Temperature = 1.0 Threshold δ = 0.4 Optimization Steps = 400 Training samples per dataset = 2000
Backbone	Qwen3-4B-Instruct-2507, Qwen3-8B
GRPO Trainer	Actor learning rate = 1×10^{-6} clip_ratio = [0.20, 0.28] Initial KL Coefficient = 1×10^{-4}
Batch Sizes	train_batch_size = 16 rollout_batch_size = 16 micro_train_batch_size = 8 micro_rollout_batch_size = 16
Lengths	Prompt Max Length = 4096 Generate Max Length = 8192
Optimizations	bf16, gradient_checkpointing, remove_padding, packing_samples, dynamic_batch_size, flash_attn

A.2 Hardware and Software Configuration

All experiments were conducted on a Linux server running Ubuntu 20.04.5 LTS. The server is equipped with 8 NVIDIA A800-SXM4 GPUs (80 GB memory per GPU) and powered by dual Intel Xeon Platinum 8358 CPUs with 1 TB system memory. The system runs CUDA 12.4, and all post-training experiments were conducted using the verl framework [38].

A.3 Details of Dataset

Dataset Overview. GRBENCH is a graph reasoning benchmark designed to evaluate how LLMs interact with large-scale, text-attributed graphs. It consists of ten real-world graphs spanning five domains: academic, e-commerce, literature, healthcare, and legal. In total, GRBENCH contains 1,740 manually designed question-answer pairs, all in English, covering three difficulty levels: easy, medium, and hard. Each question can be answered by explicitly reasoning over the corresponding domain graph rather than relying solely on parametric knowledge. All dataset statistics and construction details follow the original GRBENCH benchmark [23].

Table 4: Dataset Statistics of GRBENCH.

Domain	Topic	# Nodes	# Edges	# Temp	# Q
Academic	CS	~8M	~52M	15	150
	Biology	~4M	~39M	14	140
	Chemistry	~4M	~30M	14	140
	Material Science	~3M	~22M	14	140
	Medicine	~6M	~30M	14	140
	Physics	~2M	~33M	14	140
E-commerce	Amazon	~9M	~313M	20	200
Literature	Goodreads	~3M	~22M	24	240
Healthcare	Disease	~47K	~4M	27	270
Legal	Freelaw	~84M	~114M	18	180
SUM	–	–	–	**174**	**1740**

Academic Domain. The academic domain consists of six large bibliographic graphs covering Computer Science, Biology, Chemistry, Material Science, Medicine, and Physics. These graphs are constructed from DBLP [42] and Microsoft Academic Graph (MAG) [48]. Nodes include papers, authors, and venues, while edges represent citation relations, authorship relations, and publication venues. The academic graphs support multi-hop reasoning over citation chains, co-authorship structures, and publication histories.

E-commerce Domain. The e-commerce domain is built from the Amazon product datasets [19]. Nodes correspond to items and brands, and edges capture user behavior signals such as *also-viewed*, *also-bought*, *buy-after-viewing*, and *bought-together*. This domain emphasizes large-scale, densely connected graphs and supports aggregation-based reasoning tasks.

Literature Domain. The literature domain is based on the Goodreads dataset [46]. It contains nodes such as books, authors, publishers, and series, with edges describing authorship, publication, series membership, and book similarity. Questions in this domain often require reasoning over content attributes as well as relational structures among books and authors.

Healthcare Domain. The healthcare domain adopts the Hetionet biomedical knowledge graph [20]. It is a highly heterogeneous graph consisting of eleven node types, including diseases, compounds, genes, symptoms, and side effects. Edges describe rich

biomedical relations. This domain supports complex biomedical reasoning that involves aggregating evidence across multiple relation types.

Legal Domain. The legal domain is constructed from CourtListener data. Nodes include opinions, opinion clusters, dockets, and courts, while edges encode citation relations and hierarchical judicial structures. Legal questions often require tracing citation paths or aggregating information across multiple legal documents.

A.4 Details of Baseline

In this section, we provide detailed descriptions of all baseline methods evaluated in our experiments, including their underlying assumptions, graph interaction mechanisms, and reasoning paradigms. All baselines are implemented following their original papers and, unless otherwise specified, are evaluated under the same evaluation protocols as GraphScout to ensure fair comparison.

BaseLLM. BaseLLM serves as a non-graph baseline to assess the extent to which LLMs can answer domain-specific questions relying solely on their internal parametric knowledge. For this baseline, the model is prompted directly with the question, without access to any external graph or retrieved context.

TextRAG. TextRAG [14] treats the knowledge graph as a textual corpus by linearizing node attributes and edge information into plain text. A dense retriever is employed to fetch the most relevant text chunks based on semantic similarity with the input question, and the retrieved text is appended to the prompt for answer generation.

GraphRAG. GraphRAG [51] extends TextRAG by incorporating structural information during retrieval. Instead of retrieving isolated text chunks, GraphRAG retrieves subgraphs centered around core entities identified from the question, typically using 2-hop neighborhood expansion. The retrieved subgraphs are then linearized into text sequences and provided as context to the LLM.

Cypher. The Cypher baseline directly queries the underlying knowledge graph using Cypher. For each question, the LLM is prompted to generate a Cypher query, which is executed against the graph to retrieve relevant nodes or attributes. The query results are then returned to the LLM for answer synthesis. To adapt this method to the tasks considered in this paper, we additionally incorporate a node retrieval module to identify candidate nodes prior to Cypher query execution.

GraphCoT. GraphCoT [23] is a reasoning-based GraphRAG method that enables multi-step interaction between the LLM and the graph through a predefined set of graph traversal tools. The model incrementally explores the graph, retrieves intermediate evidence, and updates its reasoning context in a chain-of-thought manner. By decomposing complex queries into multiple reasoning steps, GraphCoT mitigates information loss caused by long linearized contexts and serves as a strong iterative reasoning baseline.

PolyG. PolyG [33] introduces an adaptive graph traversal strategy that dynamically adjusts exploration paths based on the evolving reasoning state. Unlike fixed-hop retrieval methods, PolyG allows the model to select different graph operators conditioned on the query and intermediate observations. This flexibility improves coverage for diverse question types, particularly those requiring non-uniform traversal patterns. To adapt this method to the tasks considered in this paper, we additionally incorporate a node retrieval module to identify candidate nodes prior to graph traversal.

GraphCounselor. GraphCounselor [13] employs a multi-agent collaborative framework for graph reasoning. It decomposes the reasoning process into planning, execution, and reflection stages, allowing the model to dynamically adjust reasoning depth and correct earlier mistakes through self-reflection.

LLM Backbones. To ensure robustness and generalizability, all baselines are instantiated with multiple state-of-the-art foundation models, including GPT-4o [1], GLM-4.6 [15], Qwen-Max [50], and DeepSeek-Chat [31]. Unless otherwise noted, decoding parameters (e.g., temperature and top-p) follow the default or recommended settings from the corresponding baseline implementations. In addition, for all methods that require vector retrieval, we use `Qwen3-text-embedding-v4` to obtain text embeddings.

Overall, these baselines cover a broad spectrum of graph reasoning paradigms, ranging from text-only retrieval and symbolic querying to iterative and adaptive graph exploration.

A.5 Group Relative Policy Optimization

We adopt Group Relative Policy Optimization (GRPO) [16] as the reinforcement learning algorithm for training the graph reasoning policy. GRPO is a variant of Proximal Policy Optimization (PPO) that eliminates the need for an explicit value function by estimating advantages using relative rewards within a sampled group, thereby simplifying training and reducing computational overhead.

For each input question q, a frozen reference policy π_{ref} and the current policy π_θ are used to sample a group of $|G|$ trajectories:

$$G = \{\tau_1, \tau_2, \ldots, \tau_{|G|}\}, \quad \tau_i \sim \pi_{\theta_{\text{old}}}(\cdot \mid q), \tag{7}$$

where each trajectory τ_i corresponds to a complete multi-step interaction sequence, including intermediate reasoning steps and tool invocations, and terminates with a final answer. A scalar reward r_i is assigned to each trajectory based solely on the correctness and quality of the final outcome, while intermediate steps are not explicitly supervised.

Unlike PPO, which relies on a learned value function to estimate advantages, GRPO computes a group-relative advantage by normalizing rewards within the sampled group:

$$A_i = \frac{r_i - \mu_G}{\sigma_G}, \quad \mu_G = \frac{1}{|G|}\sum_{k=1}^{|G|} r_k, \quad \sigma_G = \sqrt{\frac{1}{|G|}\sum_{k=1}^{|G|}(r_k - \mu_G)^2}. \tag{8}$$

This formulation encourages the policy to increase the likelihood of trajectories that outperform other candidates sampled for the same question, without requiring a separate critic network.

The policy is updated using a clipped surrogate objective analogous to PPO. Specifically, let

$$\rho_i(\theta) = \frac{\pi_\theta(\tau_i \mid q)}{\pi_{\theta_{\text{old}}}(\tau_i \mid q)} \tag{9}$$

denote the importance sampling ratio computed from the token-level probabilities of the generated trajectory. The GRPO objective

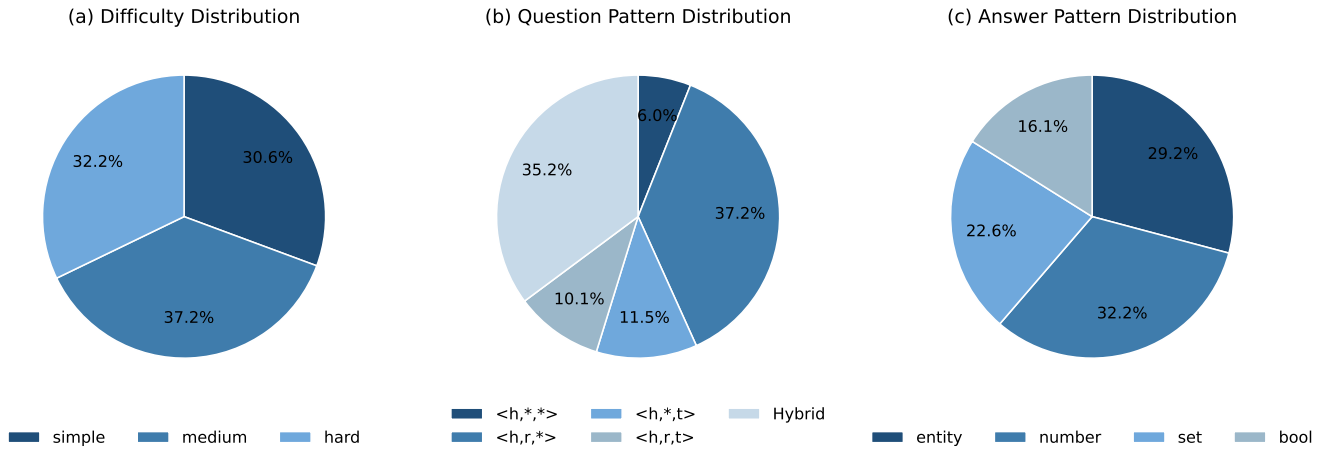

Figure 7: Distribution of the generated dataset in terms of question difficulty levels, question structural patterns, and answer types. All category labels are automatically judged by an LLM. The three pie charts summarize the proportions of different categories across the entire dataset.

is defined as:

$$L_{\text{GRPO}}(\theta) = \frac{1}{|G|}\sum_{i=1}^{|G|}\min\Big(\rho_i(\theta)A_i,\ \text{clip}\big(\rho_i(\theta), 1-\varepsilon, 1+\varepsilon\big)A_i\Big) - \beta\, D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})\,. \tag{10}$$

where ε is the clipping hyperparameter and β controls the strength of the KL regularization term. The KL penalty constrains the updated policy to remain close to the reference policy, stabilizing training and preventing excessive divergence during long-horizon reasoning.

B Additional Analysis and Experiments

B.1 Diversity Analysis of Graph Quizzer

To analyze the diversity of the data generated by Graph Quizzer, we perform an additional automatic annotation analysis on the generated questions. Specifically, during this analysis, we use DeepSeek-Chat as the judging LLM, while deliberately withholding the per-question task configurations assigned during generation, including the specific combinations of difficulty levels, question structural patterns, and answer types specified for each individual question. Instead, we only provide the definitions of these categories together with each generated question, allowing the model to independently determine the task category combination that each question actually corresponds to.

This de-conditioned annotation strategy is designed to analyze the generated data from the perspective of the questions' intrinsic semantics and structures. On the one hand, during generation, Graph Quizzer may not always strictly adhere to the predefined task instructions; on the other hand, this post hoc annotation process enables us to explicitly verify the actual task attributes reflected by each question at the semantic level.

As shown in the figure 7, the distributions of question difficulty and answer type are relatively balanced, indicating good coverage of the generated data along these dimensions. In contrast, the distribution of question structural patterns is more uneven, with hybrid questions accounting for a larger proportion. We hypothesize that this phenomenon primarily arises because, when generating questions of medium and higher difficulty, simple and single-structure question patterns are often insufficient to meet the required difficulty level, thereby encouraging the model to generate hybrid questions that incorporate multiple structural elements.

To further examine diversity beyond categorical task attributes, we analyze the generated data from a structural perspective in Figure 8. Specifically, Figure 8 presents the distributions of question token lengths and the number of clue nodes per question. The token length distribution exhibits substantial variation with a clear long-tail pattern, while the number of clue nodes spans from simple, single-clue questions to instances involving multiple reasoning cues. Together, these results indicate that Graph Quizzer generates questions with diverse linguistic realizations and varying levels of reasoning complexity.

Table 5: Proportion of failed tool calls.

	Healthcare	Literature	Academic	E-Commerce	Legal
w/o train	63.73%	67.14%	61.48%	61.15%	72.20%
w/ train	**3.14%**	**5.80%**	**5.45%**	**6.61%**	**1.27%**

B.1.1 ***Tool Invocation Reliability.*** Table 5 reports the proportion of failed tool calls before and after training across different domains. Without training, the model exhibits a high failure rate, exceeding 60% in most domains, indicating unreliable tool usage under open graph interfaces. After training with GraphScout, the failure rate drops sharply across all domains. This substantial reduction suggests that training on task-oriented exploration experience significantly improves the model's ability to invoke tools correctly and robustly, highlighting the effectiveness of GraphScout in stabilizing tool-mediated graph interaction.

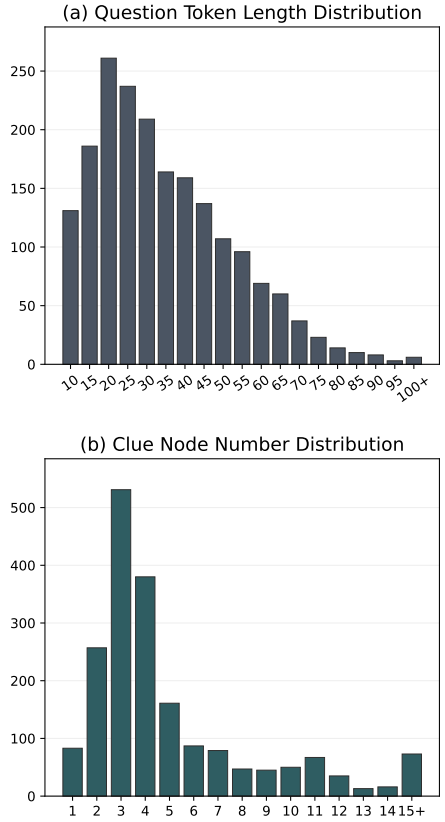

Figure 8: Distribution of (a) question token lengths and (b) the number of clue nodes per question in Healthcare dataset.

B.2 Case Study

Question: What cellular component is involved with most of the genes that are downregulated in a disease causing Aphasia, Primary Progressive?
Ground Truth: *neuron projection.*

B.2.1 ***GraphCoT Case (Failure).*** The model terminates with: *"Unfortunately, there is no information available in the graph about genes that are downregulated or associated with Primary Progressive Aphasia. Therefore, we cannot determine the cellular components involved with these genes for this disease."*

Turn 1:

```
Thought 1. The question is related to a disease node
(Aphasia, Primary Progressive) and we need to find the node
in the graph.
Action 1. RetrieveNode[Aphasia, Primary Progressive]
Observation 1. The ID of this retrieval target node is
D018888.
```

Turn 2:

```
Thought 2. The next step is to find the genes that are
downregulated by this disease.
Action 2. NeighbourCheck[D018888,
Disease-downregulates-Gene]
Observation 2. The node or neighbor type does not exist in
the graph. Please modify it.
```

Turn 3:

```
Thought 3. It seems there was an issue with the neighbor
type specified.
Action 3. RetrieveNode[Primary Progressive Aphasia]
Observation 3. The ID of this retrieval target node is
D018888.
```

Turn 4:

```
Thought 4. The node ID remains the same. Try again to find
downregulated genes.
Action 4. NeighbourCheck[D018888,
Disease-downregulates-Gene]
Observation 4. The node or neighbor type does not exist in
the graph.
```

Turn 5:

```
Thought 5. Explore alternative relations using
Disease-associates-Gene.
Action 5. NeighbourCheck[D018888, Disease-associates-Gene]
Observation 5. The node or neighbor type does not exist in
the graph.
```

Turn 6:

```
Thought 6. No associated or downregulated genes are
available for this disease.
Action 6. Finish[Unfortunately, there is no information
available in the graph about genes that are downregulated
or associated with Primary Progressive Aphasia. Therefore,
we cannot determine the cellular components involved with
these genes for this disease.]
Observation 6. Answer is INCORRECT.
```

B.2.2 ***GraphScout Case (Success).*** The model outputs: neuron projection.

Turn 1:

```
I need to break down this question step by step:

1. Find the disease "Aphasia, Primary Progressive"
2. Find which genes are downregulated in this disease
3. Find which cellular components are associated with
those genes
4. Determine which cellular component is involved with the
most genes

Let me first use the node_id_retriever to find the ID for
"Aphasia, Primary Progressive".

<tool_call>
{"name": "node_id_retriever", "arguments": {"queries":
["Aphasia, Primary Progressive"], "topk": 2}}
</tool_call>

<tool_response>
{'topk': 2, 'results': [[{'node_id': 'D018888',
'node_type': 'Symptom', 'score': 1.0, 'name': 'Aphasia,
Primary Progressive'}, {'node_id': 'D057178', 'node_type':
'Symptom', 'score': 0.79, 'name': 'Primary Progressive
Nonfluent Aphasia'}]]}
</tool_response>
```

Turn 2:

```
I see that "Aphasia, Primary Progressive" is mapped to a
Symptom node with ID `D018888`. However, I need to find
genes that are downregulated in a disease, not symptoms.

Looking at the relationship types, I can see there's
`DISEASE_DOWNREGULATES_GENE` which should connect a
disease to genes that are downregulated. But I need to
confirm if this disease is actually a disease node or if I
need to find the disease entity.

Let me check if there's a disease node with a similar name.
Maybe "Aphasia, Primary Progressive" is actually a symptom,
and I need to find the underlying disease. Alternatively,
let me first check what relationships exist from this
symptom node.

I should verify if the disease is actually a disease node
by checking the relationship types. Let me first see what
relationships this symptom node has, especially looking
for disease-related ones.

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (s:Symptom {id:
'D018888'})-[r]->(n:biomedical)\nRETURN type(r) AS
rel_type, count(DISTINCT n) AS count\nORDER BY count
DESC\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'rel_type': 'DISEASE_PRESENTS_SYMPTOM', 'count': 2}]
</tool_response>
```

Turn 3:

```
Ah! I see it now. The symptom "Aphasia, Primary
Progressive" (D018888) is presented by a disease through
the `DISEASE_PRESENTS_SYMPTOM` relationship. So I need to
find the disease that presents this symptom.

Let me find the disease node that has this symptom as a
presentation.

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (s:Symptom {id: 'D018888'})-[r:DISEA
SE_PRESENTS_SYMPTOM]->(d:Disease)\nRETURN d.id AS
disease_id, d.name AS
disease_name\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'disease_id': 'DOID:11949', 'disease_name':
'Creutzfeldt-Jakob disease'}, {'disease_id': 'DOID:10652',
'disease_name': "Alzheimer's disease"}]
</tool_response>
```

Turn 4:

```
Interesting - I see that "Aphasia, Primary Progressive" is
associated with both Creutzfeldt-Jakob disease and
Alzheimer's disease. Since the question specifically
mentions "a disease causing Aphasia, Primary Progressive",
I need to find which of these has genes downregulated.

I need to find genes that are downregulated in these
diseases. The relationship `DISEASE_DOWNREGULATES_GENE`
should connect a disease to genes that are downregulated.

Let me check if there's a `DISEASE_DOWNREGULATES_GENE`
relationship from either of these diseases to genes.

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (d:Disease {id: 'DOID:11949'})-[r:DI
SEASE_DOWNREGULATES_GENE]->(g:Gene)\nRETURN count(DISTINCT
g) AS gene_count\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'gene_count': 0}]
</tool_response>
```

Turn 5:

```
Let me try the other disease - Alzheimer's disease
(DOID:10652).

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (d:Disease {id: 'DOID:10652'})-[r:DI
SEASE_DOWNREGULATES_GENE]->(g:Gene)\nRETURN count(DISTINCT
g) AS gene_count\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'gene_count': 250}]
</tool_response>
```

Turn 6:

```
Great! So Alzheimer's disease (DOID:10652) has 250 genes
that are downregulated. Now I need to find which cellular
component is involved with most of these genes.

The relationship between genes and cellular components is
`GENE_PARTICIPATES_CELLULAR_COMPONENT`.

Let me find all the cellular components that genes
downregulated in Alzheimer's disease are associated with,
and count how many genes each component has.

I'll first find all the cellular components that are
associated with genes downregulated in Alzheimer's disease.

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (d:Disease {id: 'DOID:10652'})-[r:DI
SEASE_DOWNREGULATES_GENE]->(g:Gene)\nMATCH
(g)-[r2:GENE_PARTICIPATES_CELLULAR_COMPONENT]->(cc:Cellul
ar_Component)\nRETURN cc.id AS cc_id, cc.name AS cc_name,
count(DISTINCT g) AS gene_count\nORDER BY gene_count
DESC\nLIMIT 1\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'cc_id': 'GO:0043005', 'cc_name': 'neuron projection',
'gene_count': 35}]
</tool_response>
```

Turn 7:

```
Let me get the full list to confirm this is indeed the one
with the most genes.

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (d:Disease {id: 'DOID:10652'})-[r:DI
SEASE_DOWNREGULATES_GENE]->(g:Gene)\nMATCH
(g)-[r2:GENE_PARTICIPATES_CELLULAR_COMPONENT]->(cc:Cellul
ar_Component)\nRETURN cc.id AS cc_id, cc.name AS cc_name,
count(DISTINCT g) AS gene_count\nORDER BY gene_count
DESC\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'cc_id': 'GO:0043005', 'cc_name': 'neuron projection',
'gene_count': 35},
 {'cc_id': 'GO:0045202', 'cc_name': 'synapse',
 'gene_count': 30},
 {'cc_id': 'GO:0098796', 'cc_name': 'membrane protein
 complex', 'gene_count': 27},
 ...]
</tool_response>
```

Turn 8:

```
The result is truncated. Let me get the full result with
proper pagination or by limiting to just the top one with
the maximum count.

I'll get the exact maximum count by ensuring I get all
results properly.

<tool_call>
{"name": "code_interpreter", "arguments": {"code": "rows =
cypher(\"\"\"\nMATCH (d:Disease {id: 'DOID:10652'})-[r:DI
SEASE_DOWNREGULATES_GENE]->(g:Gene)\nMATCH
(g)-[r2:GENE_PARTICIPATES_CELLULAR_COMPONENT]->(cc:Cellul
ar_Component)\nRETURN cc.id AS cc_id, cc.name AS cc_name,
count(DISTINCT g) AS gene_count\nORDER BY gene_count
DESC\n\"\"\")\nprint(rows)"}}
</tool_call>
<tool_response>
[{'cc_id': 'GO:0043005', 'cc_name': 'neuron projection',
'gene_count': 35},
 {'cc_id': 'GO:0045202', 'cc_name': 'synapse',
 'gene_count': 30},
 ...]
</tool_response>
```

Turn 9:

```
Perfect! This confirms that the cellular component "neuron
projection" is involved with the most genes (35 genes)
among those downregulated in Alzheimer's disease, which is
the disease causing "Aphasia, Primary Progressive".

So the final answer is "neuron projection".

\answer{neuron projection}
```

B.2.3 ***Why GraphCoT fails but GraphScout succeeds.*** The failure of GraphCoT mainly stems from a *tool-to-schema mismatch* and the lack of a robust recovery strategy. Specifically, GraphCoT assumes that "Aphasia, Primary Progressive" is a `Disease` node and repeatedly queries non-existent neighbor types (e.g., `Disease-downregulates-Gene`, `Disease-associates-Gene`). Once the tool reports that the neighbor type does not exist, GraphCoT keeps retrying with the same (incorrect) schema assumption, and eventually concludes that the graph contains no relevant information, leading to an incorrect termination.

In contrast, GraphScout exhibits a *two-level advantage* that combines **reasoning** and **tool usage**. On the reasoning level, it explicitly diagnoses the entity-type mismatch (the retrieved node is a `Symptom`, not a `Disease`) and formulates a corrective plan: first resolve the underlying disease(s) that present the symptom, then query downregulated genes, and finally aggregate cellular components. On the tool level, GraphScout leverages executable Cypher queries to (i) validate available relation types from the retrieved node, (ii) traverse the graph to candidate diseases, (iii) verify which candidate disease actually contains `DISEASE_DOWNREGULATES_GENE` edges, and (iv) perform aggregation (`count(DISTINCT g)`) over `GENE_PARTICIPATES_CELLULAR_COMPONENT`. This synergy between schema-aware reasoning and executable graph operations enables GraphScout to both recover from initial ambiguity and produce

a verifiable answer, correctly identifying *neuron projection* as the most frequent cellular component among downregulated genes.

B.3 Prompt Design for Graph Quizzer

Graph Quizzer relies on a carefully structured prompting scheme to guide a strong language model to perform task-oriented yet flexible exploration over an open and programmable graph interface. Instead of using rigid question templates, the prompts specify *abstract task objectives*, including question pattern, answer type, and difficulty level, while leaving the concrete exploration strategy to the LLM. This design encourages diverse graph exploration behaviors while ensuring that the collected evidence can be reliably grounded in the underlying knowledge graph.

Base System Prompt for Graph Quizzer

You are a graph-analysis agent operating on a heterogeneous knowledge graph stored in Neo4j.
Your objective is to perform multi-round, task-oriented exploration to collect graph evidence that supports a specific question pattern, answer type, and difficulty constraint.
You interact with the graph through a programmable Python sandbox with access to a read-only Cypher interface. Your exploration should be:

- **Task-oriented**: focus on evidence relevant to the current task specification.
- **Incremental**: gather information over multiple rounds rather than exhaustively traversing the graph.
- **Verifiable**: ensure that all collected evidence can be reproduced via explicit Cypher queries.

Once sufficient evidence has been collected, you should stop exploration and construct exactly one graph reasoning question together with its answer.

B.3.1 ***Difficulty-Specific Exploration Prompts***. controls the expected depth and structural complexity of exploration, ranging from shallow, local queries to multi-hop, compositional reasoning over heterogeneous graph structures. It determines the allowed traversal depth, the number of intermediate entities, and whether hybrid reasoning patterns are permitted.

Difficulty Level: SIMPLE

Design Goal. Encourage shallow, interpretable graph exploration that can be resolved with minimal reasoning steps.
Exploration Constraints.

- Restrict traversal depth to 1–2 hops.
- Prefer a single dominant relationship type.
- Avoid compositional or hybrid reasoning patterns.

Evidence Scope. Collected evidence should support direct fact retrieval or simple aggregation, and be sufficient to answer the question without multi-step inference.
Typical Use Cases. Single-hop entity lookup, attribute retrieval, or basic neighborhood queries.

Difficulty Level: MEDIUM

Design Goal. Promote controlled multi-step reasoning while maintaining clear semantic structure.
Exploration Constraints.

- Explore 2–4 hop paths with explicit intermediate nodes.
- Involve at least two distinct relationship types or node types.
- Allow limited use of filtering, ranking, or property constraints.

Evidence Scope. The collected evidence should form a clear reasoning chain that can be verbalized into a natural-language question.
Typical Use Cases. Multi-hop entity discovery, constrained aggregation, or relational queries requiring intermediate reasoning.

Difficulty Level: HARD

Design Goal. Encourage non-trivial, compositional graph reasoning that cannot be resolved by local lookup.
Exploration Constraints.

- Allow hybrid or combined question patterns.
- Explore 3–5 hop heterogeneous paths.
- Require traversal through multiple intermediate entities and relation types.
- Explicitly avoid staying within a single entity's immediate neighborhood.

Evidence Scope. Evidence must support multi-step dependency between reasoning stages, ensuring that the final answer depends on earlier intermediate results.
Typical Use Cases. Recommendation-style reasoning, cross-entity synthesis, and complex relational verification.

B.3.2 ***Query Pattern-Specific Exploration Prompts***. specifies the abstract structural form of a graph question using a (possibly incomplete) triple representation. Rather than fixing concrete templates, the pattern defines which components are given and which

must be discovered through exploration, thereby guiding the direction and scope of evidence collection.

Pattern: $\langle h, _, _ \rangle$ (Entity-Centric Exploration)

Task Definition. Only the subject entity h is given, while both the predicate and object are unspecified.
Exploration Objective. Collect broad but structured information about the subject entity, including:

- the node type and key attributes of h,
- dominant outgoing relation types,
- representative neighboring entities and their types,
- simple graph statistics.

Design Constraint. Exploration should remain local and avoid aggregating excessive or irrelevant neighborhoods.

Pattern: $\langle h, r, * \rangle$ (Object-Finding)

Task Definition. The subject entity h and predicate r are specified, while the object is unknown.
Exploration Objective. Identify target entities that satisfy the given relation(s), possibly involving:

- single-hop or multi-hop predicate chains,
- filtering or aggregation constraints,
- explicit intermediate nodes as supporting evidence.

Design Constraint. The exploration must yield a reproducible query result and avoid ambiguous answer sets.

Pattern: $\langle h, *, t \rangle$ (Relationship Discovery)

Task Definition. Both subject h and object t are given, while the relationship between them is unknown.
Exploration Objective. Discover and explain how h and t are connected in the graph by:

- enumerating valid relation paths,
- identifying intermediate entities,
- selecting concise and interpretable paths as evidence.

Design Constraint. Preference is given to short or semantically meaningful paths rather than exhaustive enumeration.

Pattern: $\langle h, r, t \rangle$ (Verification)

Task Definition. Subject, predicate, and object are all specified.
Exploration Objective. Verify whether the specified relationship (or constrained relation chain) holds in the graph.
Evidence Requirement. The final answer must be derived from an explicit verification query (e.g., existence check or count-based test) and supported by at least one concrete witness when the result is positive.

Hybrid Pattern (Compositional Graph Reasoning)

Task Definition. Combine exactly two different question patterns into a single coherent question, where the second reasoning step depends on the result of the first.
Exploration Objective. Ensure that:

- both reasoning steps are necessary to answer the question,
- intermediate results from the first step are explicitly reused,
- the final question cannot be answered by a single local lookup.

B.3.3 ***Answer-Type-Specific Prompt Constraints.*** imposes explicit constraints on the form and computability of the final answer, including entity, boolean, numeric, and set-valued outputs. Each answer type is associated with verifiable evidence requirements (e.g., uniqueness, existence, aggregation), ensuring that generated questions admit deterministic, graph-grounded supervision.

Answer Type: ENTITY

Objective. Produce exactly one uniquely identifiable real-world entity as the answer.
Hard Constraints.

- The candidate answer set must be reduced to exactly one entity.
- The model must verify uniqueness using an explicit Cypher aggregation.
- Internal node identifiers are forbidden; only human-readable names are allowed.

Answer Type: BOOLEAN

Objective. Verify whether a specified relationship or constraint holds in the graph.
Hard Constraints.

- The boolean value must be derived from an explicit verification query (e.g., existence check or count-based test).
- TRUE is returned iff the computed count is strictly greater than zero.
- At least one concrete witness must be identified when the result is TRUE.

Answer Type: NUMBER

Objective. Return a single numeric value computed from the graph.
Hard Constraints.

- The number must be produced by a Cypher aggregation query returning exactly one row.
- LIMIT clauses are forbidden in final counting queries.

Answer Type: SET

Objective. Return a small, interpretable set of entities.
Hard Constraints.

- The total set size must be computed explicitly.
- If the set size exceeds a predefined threshold (e.g., 10), additional constraints must be introduced.
- The final answer must be a comma-separated list of entity names.

B.4 Prompt Design for Graph Solver

B.4.1 ***Tool Interface Formatting.*** We adopt the Qwen3 chat template, which automatically injects tool descriptions into the model prompt at runtime. When a non-empty tool set is provided, the template constructs a system-level instruction that appends a dedicated Tools block containing the JSON schemas of all available tools. This block is enclosed by $< tools > \ldots < /tools >$ tags and precedes the dialogue history. The same system message also specifies a strict tool invocation protocol, requiring the model to emit function calls as structured JSON objects wrapped by $< tool_call > \ldots < /tool_call >$, and received by $< tool_response > \ldots < /tool_response >$. As a result, both tool semantics and calling conventions are explicitly grounded in the prompt, enabling deterministic parsing and execution of tool calls during inference.

Code Interpreter Tool Description

tool_schema:
type: "function"
function:
name: "code_interpreter"
description: "Execute Python code for querying and analyzing a heterogeneous graph stored in Neo4j. Inside the executed Python, use the provided synchronous function 'cypher(query, params=None, limit=None)' (not a separate tool) to run read-only Cypher queries."
parameters:
type: "object"
properties:
code:
type: "string"
description: "Python code"
required: ["code"]

Node Retriever Tool Description

tool_schema:
type: "function"
function:
name: "node_id_retriever"
description: "Retrieve top nodes ids from a heterogeneous graph by semantic similarity (batch). You can use this tool to get target nodes ids first and then continue next steps."
parameters:
type: "object"
properties:
queries:
type: "array"
description: "A list of query strings. Such as ['Nausea', 'MTUS1']"
items:
type: "string"
topk:
type: "integer"
description: "Number of retrieved nodes per query. Default is 2."
default: 2
required: ["queries"]

B.4.2 ***Formatting of Graph Descriptions.*** To ensure clarity and reproducibility, we explicitly standardize the formatting of graph descriptions and code examples in the Graph Solver.

System Prompt

You are given a < *domain_label* > heterogeneous knowledge graph stored in Neo4j.
- Backend: Neo4j (Cypher)
- Label scope: :< *domain_label* >
- Node properties:
  - The dataset has these node_types: < *node_type_list* >
  - each node_type has its own set of type-specific properties: < *properties_of_each_type* >
  - There are several types of edges (relationship type) in this graph: < *edge_type_list* >
- How to query in code: < *code_examples* >

- Output rule: The very last line of your response must be exactly in the form `\answer{FINAL_ANSWER}`. Do not include any extra text on that last line.