GraphWalk: Enabling Reasoning in Large Language Models through Tool-Based Graph Navigation

Anonymous ACL submission

Abstract

Large Language Models struggle with complex queries over enterprise-scale knowledge graphs due to context limits and unreliable retrieval methods. We present GraphWalk, a tool-based framework that enables reasoning for LLMs through sequential graph navigation. Instead of relying on implicit prompts, GraphWalk forces explicit reasoning by strategically choosing fundamental graph operation tools to collect the necessary evidence to answer queries. Each tool call represents a verifiable reasoning step, creating a transparent execution trace. To isolate structural reasoning from world knowledge, we evaluate on entirely synthetic graphs with random values and labels. Our benchmark spans 12 query templates from basic retrieval to complex logical operations. Results demonstrate that GraphWalk successfully transforms language models into systematic reasoning agents, handling first-order logic constructs that traditional approaches fail to manage reliably.

1 Introduction

Recent advancements have seen Large Language Models (LLMs) achieve state-of-the-art performance on challenging reasoning benchmarks such as MMLU (Hendrycks et al., 2021) and GSM8K (Cobbe et al., 2021). However, a significant gap persists in aligning these models with enterprise-specific knowledge graphs (KGs), which are too large for in-context processing and whose complex query patterns often defy standard retrieval methods. Current approaches remain insufficient. Graph-RAG (Edge et al., 2024), subgraph extraction methods (Zhang et al., 2022), and in-context graphs (Fatemi et al., 2024; He et al., 2024) are inherently lossy, removing crucial structural context. Text-to-query generation approaches (Feng et al., 2024; D'Abramo et al., 2025; Steinigen et al., 2024; Zhong et al., 2025) frequently produce erroneous queries, trapping models in unproductive self-correction loops. We reframe this problem from in-context understanding to interactive exploration. Our framework positions the LLM as an autonomous agent equipped with fundamental graph operation tools. The agent navigates the graph by executing these tools sequentially to gather information for answering questions. This tool-based interaction forces an explicit chain of thought, unlike "think step-by-step" prompts that LLMs may ignore, our framework mandates a verifiable sequence of actions where each tool call represents a discrete reasoning step, creating a transparent execution trace. Our contributions are:

1. *A Framework for Autonomous Graph Navigation:* We introduce a tool-based agentic framework enabling LLMs to reason over arbitrarily large KGs through sequential exploration.

2. *Controlled Synthetic Graph Benchmark:* We design a configurable synthetic graph generator that produces entirely random graphs with arbitrary values, along with a benchmark of 12 query templates spanning multiple complexity levels to isolate and evaluate pure structural reasoning capabilities.

2 Related Work

The effort to synergize LLMs and KGs has resulted in several distinct approaches:

2.1 Graph Textualization and RAG:

Early approaches serialize graph structures into text through triple linearization or JSON representations for injection into LLM context windows. Benchmarks including GrailQA (Gu et al., 2021), GrailQA++ (Dutt et al., 2023), and WebQSP (Yih et al., 2016) evaluate question-answering over KG contexts. These Retrieval-Augmented Generation methods focus on retrieving and formatting relevant subgraphs (Edge et al., 2024; Zhang et al., 2022), but struggle with scalability and preserving global graph structure.

2.2 LLMs as Query Generators:

These approaches train LLMs to translate natural language questions into formal query languages like SPARQL or Cypher (Kovriguina et al., 2023; Zahera et al., 2024; D'Abramo et al., 2025; Ozsoy et al., 2025). While leveraging structured query engines, they frame tasks as one-shot translation problems unsuitable for exploratory reasoning. Query errors often result in correction loops without resolution.

2.3 Hybrid LLM-GNN Architectures:

Integration of Graph Neural Networks with LLMs (Mavromatis and Karypis, 2025; Liu et al., 2025) uses GNNs to learn topology-aware node and edge embeddings, providing compressed structural representations. This enriches LLM understanding through improved internal representations rather than explicit, verifiable tool-use behavior.

2.4 Tool-Augmented Graph Agents:

Our work aligns with the emerging "LLM as Agent" paradigm applied to graphs. Frameworks like `Reason-Align-Respond` (RAR) (Shen et al., 2025) and `SubgraphRAG` (Li et al., 2025) demonstrate potential for breaking down complex queries through interactive paradigms where LLMs actively participate rather than passively read. We extend this approach with critical methodological distinctions for rigorous, unbiased reasoning evaluation.

3 Methodology: Reasoning by Navigation on Synthetic Graphs

Our proposed solution centers on an agentic framework where an LLM navigates a property graph to gather information. The only provided information are the graph schema, and the tools by which the agent is able to explore the graph.

3.1 Isolating Structural Reasoning from Parametric Knowledge

A primary contribution of our methodology is the use of **entirely random graphs**. In contrast to benchmarks based on factual domains like Freebase (Bollacker et al., 2008) or Wikidata (Vrandečić and Krötzsch, 2014), these graphs are synthetically generated with meaningless labels for nodes, properties, and relationships.

This design is intentional and crucial as it prevents the LLM from using its vast internal world knowledge to infer connections or "guess" answers. The LLM must rely exclusively on the provided schema and the results of its tool-based exploration. This allows us to address a fundamental confound in existing KGQA evaluations and purely assess the agent's capacity for logical reasoning.

3.2 The Agent's Toolset and Reasoning Loop

The LLM agent is provided with a minimal yet sufficient set of tools for graph traversal. This tool suite is designed to be orthogonal, forcing the agent to compose simple actions to perform complex tasks. The core tools include:

- `get_node_by_property`: Retrieves nodes of a specified label that match a given property-value pair. This is the primary entry-point tool for locating specific entities in the graph when their identifying property (e.g., name, ID, or unique attribute) is known.
- `get_all_nearest_neighbors`: Returns all nodes directly connected to a specified node through any relationship type. This tool enables exploration of a node's immediate neighborhood and discovery of its direct connections.
- `get_unique_property_values`: Retrieves all distinct values for a specified property across nodes or relationships of a given type. This tool enables discovery of available entities and validation of specific values before searching, serving as a critical data exploration mechanism when the agent needs to enumerate possible options or understand the scope of information available for a particular property.
- `think`: Records intermediate reasoning steps by taking a string input and returning it unchanged, creating a visible record in the agent's execution trace. This tool enables the agent to document its thought process, plan multi-step approaches, and articulate strategic decisions without querying the graph.

Each tool (except for the `think` tool) is equivalent to a cypher query which is executed on the graph database. Given a question, the LLM selects tool(s) to *act*, and then *observes* the result returned from a deterministic graph executor. This cycle

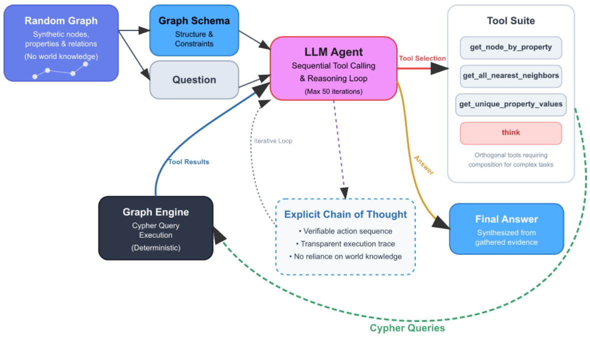

Figure 1: An overview of the agentic framework.

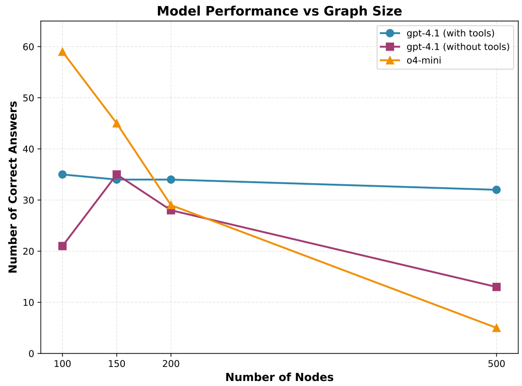

Figure 2: Effect of graph size on model performance. The number of correct answers (out of 120 total) decreases as graph complexity increases for all models. Tool-equipped models demonstrate better robustness to increasing graph size compared to models without tool access.

repeats until the agent decides that it has gathered sufficient evidence to formulate a final answer, or a limit of 30 iterations is reached. Figure 1 demonstrates an overview of our framework. More details regarding the agent tools are laid out in appendix B.

3.3 The Agent's Reasoning Tasks

To evaluate the agent's reasoning capabilities, we designed a benchmark of 12 query templates, each targeting a distinct aspect of graph reasoning. These templates are grouped into three main categories, described below.

3.3.1 Retrieval and Aggregation:

This class includes direct lookups and summarization tasks. The `node_by_property_query` asks the agent to find all nodes of a specific type with a property matching a given value, while `relationship_by_property_query` requires finding all relationships of a certain type where a property matches a value, returning the connected nodes. Aggregation tasks such as `node_count_query` and `relationship_count_query` challenge the agent to count nodes or relationships matching certain criteria, and `node_with_most_relationships` asks for the node with the highest number of outgoing relationships of a specific type.

3.3.2 Path and Relational Traversal:

These queries test the agent's ability to perform multi-hop reasoning and follow connections across the graph. The `path_finding_query` involves finding all paths connecting a source node type to a target node type through a specified intermediate node type. The `variable_hop_path_query` is more challenging, requiring the agent to find paths of variable length between two node types, followed by an additional step to any other node. The `path_from_specific_node_query` asks which nodes of a target type can be reached within a specific number of hops from a given starting node, and the `remote_node_property_query` requires finding a node reachable in two or more hops (but not directly) and returning one of its properties.

3.3.3 Complex Logical Composition.

These queries mirror constructs from first-order logic, testing the agent's ability to handle conjunctions and negation. The `compositional_intersection_query` requires identifying nodes that satisfy two independent relational conditions simultaneously, equivalent to a logical AND operation, $(\exists y\, R(x, y)) \wedge (\exists z\, S(x, z))$. Negation is tested with `negation_with_connection_query`, which asks for nodes connected to a "positive" target type but not to a "negative" target type, corresponding to $(\exists y\, P(x, y)) \wedge \neg(\exists z\, Q(x, z))$, and with `negation_on_rel_property_query`, which requires finding nodes connected by a specific relationship type where a property on that relationship does not equal a certain value.

The exact templates and parameters for all 12 query types are detailed in appendix C.

4 Experimental Setup

4.1 Baseline Evaluation

We evaluated seven language models: gpt-4.1-nano, gpt-4.1-mini, gpt-4.1 (OpenAI, 2025a), gpt-4o-mini, gpt-4o (OpenAI, 2024), o3-mini, and o4-mini (OpenAI, 2025b) (with default reasoning effort for the reasoning models) across our benchmark of

Model	Tools	Correct	Accuracy	Precision	Recall	F1	False Positives	Tool Calls
gpt-4o-mini	True	18	15.00	0.23	0.25	0.23	170	7362
	False	6	5.00	0.35	0.24	0.26	350	-
gpt-4o	True	34	28.33	0.42	0.39	0.39	235	2548
	False	18	15.00	0.50	0.44	0.44	300	-
gpt-4.1	True	**35**	29.17	0.47	0.41	0.43	232	2483
	False	21	17.50	0.52	0.49	0.48	1006	-
gpt-4.1-mini	True	32	26.67	0.49	0.43	0.43	226	3561
	False	20	17.7	0.49	0.48	0.46	522	-
gpt-4.1-nano	True	11	9.17	0.16	0.14	0.14	105	1065
	False	7	5.83	0.33	0.29	0.27	522	-
o3-mini	False	27	22.50	0.55	0.47	0.49	195	-
o4-mini	False	**59**	49.17	0.68	0.67	0.66	248	-

Table 1: Performance of LLMs on the query set. "Correct" refers to the total number of correct answers (out of 12 questions and out of 10 runs, 120 in total). The reasoning models are used without tools to provide a baseline. "Tools" refers to tool-use or lack thereof. Accuracy, Precision, Recall, and F1 scores are average scores across the 120 runs. "False Positives" refers to the total number of false positives for a given model. "Tool Calls" refers to the total number of tool calls. Highest number of correct results for a non-reasoning model and a reasoning model are highlighted in boldface.

Model	Metric	100	150	200	500
gpt-4.1 (with tools)	Acc.	29.17	28.33	28.33	**26.67**
	P.	0.47	0.39	0.40	**0.39**
	R.	0.41	0.37	0.38	**0.35**
	F1	0.43	0.37	0.38	**0.35**
	F.P.	232	149	170	151
gpt-4.1 (without tools)	Acc.	17.5	29.17	23.33	10.83
	P.	0.52	0.43	0.48	0.37
	R.	0.49	0.49	0.50	0.33
	F1	0.48	0.43	0.46	0.31
	F.P.	1006	1182	800	957
o4-mini	Acc.	49.17	37.5	24.17	4.17
	P.	0.68	0.56	0.40	0.13
	R.	0.67	0.51	0.33	0.08
	F1	0.66	0.52	0.35	0.09
	F.P.	248	108	44	**108**

Table 2: Effect of graph size on model performance across different node counts. Metrics are reported for gpt-4.1 with and without tool access, and o4-mini without tools. Acc., P., R., F1, and F.P. refer to average Accuracy, Precision, Recall, F1 scores, and the total number of False Positives, respectively. Best results with 500 nodes are highlighted in boldface.

12 query templates [1]. The experiments were conducted on a remote server equipped with dual Intel Xeon Platinum 8168 processors, each featuring 20 physical cores and 40 logical CPUs in total. The server architecture supports full virtualization, running a Linux environment with Neo4j (Community Edition) (Eifrem et al., 2016) deployed [2]. To ensure statistical robustness, each model was tested on 10 independently generated graph instances, with results averaged across all runs. Each graph instance is an entirely different and random graph, and for each graph the question templates are filled with entities picked randomly in that particular graph. This yields a total of 120 queries per model (12 templates × 10 runs). We ensure that for each question, an answer definitely exists in the graph. In these synthetic graphs, all labels, node types, relationship classes, property keys, and values are generated as random 4-8 character strings verified against an English dictionary to ensure non-semantic content. Models are evaluated in two configurations-with tool access (schema + toolset provided) and without tools (full graph serialized in context). More details regarding the representation of the schema and the prompt used are given in appendix A.

4.2 Graph Configuration for Primary Experiments

The initial experimental configuration (table 1) utilized graphs with controlled structural complexity: a maximum of 100 nodes distributed across possible 4 node classes and 2 relationship classes. Each node contained an average of 3 properties, while relationships similarly maintained 3 properties on average. Property values were drawn from a pool of 5 possible values per property, with all labels generated as random character strings of 4-8 characters in length.

4.3 Scaling Experiments

To assess model robustness under increasing complexity, we conducted a second set of experiments (table 2 and figure 2) where both graph size and structural complexity were increased. Graph sizes

[1]The OpenAI models used in this work are proprietary. No redistribution or repackaging of these models is included.

[2]Neo4j Enterprise Edition is proprietary software licensed under commercial terms by Neo4j, Inc. The Community Edition is licensed under GPLv3.

were scaled to 150, 200, and 500 nodes. Concurrently, we enriched the schema complexity: node types expanded to 8 classes, relationship types to 4 classes, with average properties per node and relationship both increasing to 6. The number of possible values per property doubled to 10, while label length constraints remained consistent at 4-8 characters.

4.4 Output Format Handling and Evaluation

A recurring challenge observed across models was non-compliance with the specified JSON output format. Models occasionally produced correct answers embedded within plain text rather than adhering to the structured format required for automated evaluation. To ensure comprehensive assessment of model capabilities, we implemented a two-stage extraction process. When standard JSON parsing failed, we employed an LLM-based extractor (powered by gpt-4o-mini) to extract the substantive answer from the model's output.

4.5 Scope and Implications

It is important to note that our benchmark of 12 query templates, while diverse in complexity, is not exhaustive of all possible graph reasoning tasks. Rather, this curated set serves to demonstrate a fundamental principle: that equipping LLMs with minimal, orthogonal tools for graph traversal can noticeably improve performance on structural reasoning tasks. The relatively modest performance achieved even on this focused benchmark, where models struggle with queries that are answerable through basic graph operations, highlights the substantial gap between current LLM capabilities and reliable graph reasoning. These results reveal significant room for advancement in this domain and suggest that graph-based reasoning remains a frontier challenge for language models. The fact that models equipped with explicit navigation tools demonstrate more consistent performance than those relying on implicit reasoning suggests compelling evidence for the necessity of structured, tool-mediated approaches to graph query answering in production systems.

5 Discussion

5.1 The Baseline Evaluations

Table 1 reveals distinct performance patterns across model configurations. Among non-reasoning models equipped with tools, gpt-4.1 achieves the highest performance, while gpt-4o-mini without tool access demonstrates the weakest results. As expected, o4-mini, a reasoning model, outperforms all other configurations, which is attributable to both its architectural design for multi-step reasoning and the fact that the 100-node graphs remain fully within its context window.

The most striking finding is the consistent performance gain when models use graph traversal tools. Every model shows substantial improvements in accuracy, precision, recall, and F1 scores with tool access, demonstrating that explicit graph navigation provides more reliable reasoning than pattern matching over serialized representations. Tool-equipped models also exhibit dramatically lower false positive rates, even surpassing reasoning models o3-mini and o4-mini. This suggests deterministic tool execution potentially constrains hallucination by enforcing factual grounding through explicit database queries rather than pattern-based inference.

Another noteworthy observation is that gpt-4.1 with tool access outperforms o3-mini, despite the latter being architecturally optimized for reasoning tasks. This result underscores a fundamental principle: structured access to information through specialized tools can compensate for, and in some cases exceed implicit reasoning capabilities when the task involves verifiable operations over structured data.

5.2 Experiments with Graph Size

Table 2 and Figure 2 present results from our scalability experiments, in which we tested gpt-4.1 (the best-performing non-reasoning model) and o4-mini across increasing graph sizes. The findings reveal a critical advantage of tool-based navigation: gpt-4.1 with tools maintains relatively stable performance across all graph sizes. In contrast, both gpt-4.1 without tools and o4-mini demonstrate substantial performance decline as graph complexity increases.

The degradation is particularly dramatic for o4-mini, which drops from 59 correct answers at 100 nodes to merely 5 at 500 nodes-despite the graph still fitting within its context window. This catastrophic failure at scale demonstrates a fundamental limitation of in-context approaches: simply having sufficient context capacity does not guarantee effective reasoning over large structured data. The 500-node graph, (which barely fits within o4-mini's context window according to our token measurements) exposes the inability of even reasoning-specialized models to maintain coherent graph traversal when processing entire serialized structures.

5.3 Model Limitations and Failure Modes

Our evaluation revealed several recurring failure modes that illuminate the specific challenges LLMs face when reasoning over graph structures. Without tool access, models consistently demonstrated incomplete exploration patterns, often fixating on the first few relationships of a node and abandoning searches when answers were not immediately apparent, even when correct information existed within the same node's connections in the serialized context. Additionally, models frequently failed to correctly ground their reasoning by selecting inappropriate starting nodes, leading to hallucinated entry points that derailed entire reasoning chains.

Tool-based agents, while substantially more effective overall, exhibited their own characteristic failures. Across both settings, we observed a persistent "last mile" problem: models would successfully gather correct evidence but fail to format responses according to the specified JSON schema, instead providing conversational answers that demonstrated correct reasoning but violated output requirements. These observations underscore both the value of structured tool access for graph navigation and the continued challenges in strategic replanning and instruction adherence.

Conclusion

GraphWalk demonstrates that tool-based graph navigation significantly improves LLM performance on complex graph reasoning tasks, as shown in table 1. The graph consists of 100 nodes in each experiment. This has been done in order for the graph to fit inside the context window of the LLMs when testing without tools, and to facilitate better comparison between the two settings. By forcing sequential tool selection, our framework transforms implicit reasoning into explicit, verifiable action sequences. This approach enhances accuracy while providing transparency: each tool call represents a discrete, inspectable reasoning step. Our results show that non-reasoning LLMs can achieve systematic graph exploration when equipped with appropriate tools, offering a promising direction for reliable knowledge graph question answering.

This work demonstrates that tool-based graph traversal provides a robust and scalable ground for LLM-driven graph reasoning. Our experimental results establish three key findings. First, equipping language models with minimal, orthogonal graph operation tools yields consistent and substantial performance improvements across all model families tested. Second, by evaluating on synthetic graphs with random labels, we isolated pure structural reasoning from world knowledge, revealing that current LLMs struggle significantly with fundamental graph traversal tasks when they cannot leverage learned semantic associations-exposing considerable room for advancement in this domain.

Most compellingly, our scalability experiments demonstrate a critical advantage of tool-mediated approaches: while even the best-performing reasoning model (o4-mini) suffers performance degradation at larger graph sizes, tool-equipped models maintain stable performance across all scales tested. This finding establishes that explicit, verifiable tool execution provides superior reliability compared to implicit in-context processing.

GraphWalk establishes a principled framework for graph question answering that prioritizes transparency, scalability, and factual grounding-qualities essential for enterprise deployment. Our results validate that structured tool access can enhance and, in some cases, exceed the capabilities of reasoning-specialized models when tasks involve verifiable operations over structured data.

Ethical Considerations

The GraphWalk framework promotes transparency in LLM reasoning by generating an explicit, auditable trace of tool calls, grounding its conclusions in verifiable data from the knowledge graph. This enhances accountability compared to opaque, in-context reasoning.

The primary ethical risk is that the framework's reliability is entirely dependent on the integrity of the underlying knowledge graph. If deployed on a graph containing biased or inaccurate information, GraphWalk could efficiently surface and compose these flaws into seemingly credible, yet harmful or false, conclusions. The transparent reasoning path could lend an undeserved sense of authority to these outputs. Therefore, any real-world application of this framework necessitates careful governance of the underlying data and scrutiny of the tool-based reasoning, especially in sensitive domains.

Limitations

While our 12-query benchmark successfully demonstrates the efficacy of tool-based graph navigation, it represents a subset of possible graph reasoning tasks. A more comprehensive evaluation suite encompassing additional query patterns would provide deeper insights into the boundaries of tool-augmented reasoning capabilities. Also, our evaluation focuses on OpenAI's model family. However, extending this analysis to open-source models (e.g., Llama, Mistral, Qwen) and alternative closed-source offerings (e.g., Claude, Gemini) would strengthen generalizability claims and reveal whether tool-based advantages persist across diverse architectural approaches and training paradigms.

Moreover, our minimal toolset intentionally tests whether basic primitives suffice for complex reasoning. Future work could explore richer tool configurations, including aggregate functions, subgraph extraction operators, and specialized traversal patterns (e.g., shortest path, centrality measures). Investigating the trade-off between tool granularity and reasoning efficiency remains an open question.

This work establishes tool-based navigation as a viable approach but does not directly compare against existing graph-LLM integration methods such as GraphRAG, subgraph retrieval systems, or text-to-Cypher generation approaches. This evaluation would contextualize our contributions within the broader landscape of graph reasoning solutions and identify complementary strengths across methodologies.

References

Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1247–1250.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Jacopo D'Abramo, Andrea Zugarini, and Paolo Torroni. 2025. Investigating large language models for text-to-SPARQL generation. In *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, pages 66–80, Albuquerque, New Mexico, USA. Association for Computational Linguistics.

Ritam Dutt, Sopan Khosla, Vinayshekhar Bannihatti Kumar, and Rashmi Gangadharaiah. 2023. GrailQA++: A challenging zero-shot benchmark for knowledge base question answering. In *Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 897–909, Nusa Dua, Bali. Association for Computational Linguistics.

Jacopo D'Abramo, Andrea Zugarini, and Paolo Torroni. 2025. Investigating large language models for text-to-sparql generation. In *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, pages 66–80.

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*.

Emil Eifrem, Johan Svensson, and Peter Neubauer. 2016. Neo4j: The graph database. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data (SIGMOD '16), Demo Track*. ACM. Software available at `https://neo4j.com/`.

Bahare Fatemi, Jonathan Halcrow, and Bryan Perozzi. 2024. Talk like a graph: Encoding graphs for large language models. In *The Twelfth International Conference on Learning Representations*.

Yanlin Feng, Simone Papicchio, and Sajjadur Rahman. 2024. Cypherbench: Towards precise retrieval over full-scale modern knowledge graphs in the llm era. *arXiv preprint arXiv:2412.18702*.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. 2021. Beyond i.i.d.: Three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, pages 3477–3488, Ljubljana, Slovenia. ACM.

Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. In *Advances in Neural Information Processing Systems*, volume 37.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *International Conference on Learning Representations (ICLR)*.

Liubov Kovriguina, Roman Teucher, Daniil Radyush, and Dmitry Mouromtsev. 2023. Sparqlgen: One-shot prompt-based approach for sparql query generation. In *Proceedings of the Posters and Demos Track of the 19th International Conference on Semantic Systems (SEMANTiCS 2023)*, volume 3526 of *CEUR Workshop Proceedings*.

Mufei Li, Siqi Miao, and Pan Li. 2025. Simple is effective: The roles of graphs and large language models in knowledge-graph-based retrieval-augmented generation. In *Proceedings of the Thirteenth International Conference on Learning Representations*.

Guangyi Liu, Yongqi Zhang, Yong Li, and Quanming Yao. 2025. Dual reasoning: A gnn-llm collaborative framework for knowledge graph question answering. In *Proceedings of the Second Conference on Parsimony and Learning*.

Costas Mavromatis and George Karypis. 2025. GNN-RAG: Graph neural retrieval for efficient large language model reasoning on knowledge graphs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 16682–16699, Vienna, Austria. Association for Computational Linguistics.

OpenAI. 2024. GPT-4o system card. *Preprint*, arXiv:2410.21276.

OpenAI. 2025a. Introducing GPT-4.1 in the API. `https://openai.com/index/gpt-4-1/`. Accessed: October 2025.

OpenAI. 2025b. Introducing OpenAI o3 and o4-mini. `https://openai.com/index/introducing-o3-and-o4-mini/`. Accessed: October 2025.

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

Xiangqing Shen, Fanfan Wang, and Rui Xia. 2025. Reason-align-respond: Aligning llm reasoning with knowledge graphs for kgqa. *Preprint*, arXiv:2505.20971.

Daniel Steinigen, Roman Teucher, Timm Heine Ruland, Max Rudat, Nicolas Flores-Herr, Peter Fischer, Nikola Milosevic, Christopher Schymura, and Angelo Ziletti. 2024. Fact finder–enhancing domain expertise of large language models by incorporating knowledge graphs. *arXiv preprint arXiv:2408.03010*.

Denny Vrandečić and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Communications of the ACM*, 57(10):78–85.

Wen-tau Yih, Matthew Richardson, Christopher Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206, Berlin, Germany. Association for Computational Linguistics.

Hamada Mohamed Abdelsamee Zahera, Manzoor Ali, Mohamed Ahmed Sherif, Diego Moussallem, and Axel-Cyrille Ngonga Ngomo. 2024. Generating sparql from natural language using chain-of-thoughts prompting. In *Proceedings of the 20th International Conference on Semantic Systems (SEMANTiCS 2024)*, Amsterdam, Netherlands.

Xikun Zhang, Antoine Bosselut, Michihiro Yasunaga, Hongyu Ren, Percy Liang, Christopher D Manning, and Jure Leskovec. 2022. Greaselm: Graph reasoning enhanced language models for question answering. In *International Conference on Learning Representations (ICLR)*.

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

A Appendix: Graph Schema and The Prompt

In this section, we provide detailed information about our proposed framework. We present specifics on an example of a complete graph schema, a comprehensive description of the agent's tools, and the full templates for the experimental questions used in our benchmark.

A.1 An Example of the Graph Schema

The graph schema is the only structural information provided to the LLM agent at the beginning of each task. It defines the blueprint of the graph, outlining all possible node types, relationship classes, and their associated properties. This information is critical, as the agent must use it to formulate valid traversal and query plans.

The schema is presented to the agent in a structured format that details each entity's type, name, Cypher representation, and available properties. As all names are randomly generated, the agent cannot infer any semantic meaning and must rely entirely on this structural definition.

Table 3 provides a representative example of the schema for one of the randomly generated graphs used in our experiments, which contained

100 nodes. This demonstrates the non-semantic nature of the labels and the structure the agent must interpret.

#	Entity Type	Entity Name	Cypher Pattern	Property
0	Node	Cevaz	(:Cevaz)	bexame
1	Node	Cevaz	(:Cevaz)	key
2	Node	Cevaz	(:Cevaz)	tanu
...	...	...	...	...
18	Relationship	EPUQOSS	(:Cevaz)-[:EPUQOSS]->(:Egodpw)	ukog
19	Relationship	EPUQOSS	(:Cevaz)-[:EPUQOSS]->(:Egodpw)	uqpc
...	...	...	...	...
34	Relationship	LAJOZOS	(:Cevaz)-[:LAJOZOS]->(:Cevaz)	bzle
35	Relationship	LAJOZOS	(:Cevaz)-[:LAJOZOS]->(:Cevaz)	uhiro
...	...	...	...	...

Table 3: A truncated example of the schema provided to the agent for a 100-node graph. The schema lists all node types, relationship types, and their respective properties.

A.2 Prompt Structure

The agent's reasoning process is initiated with a structured prompt designed for clarity and efficiency. The prompt consists of three core components: the graph schema and the natural language question. This minimalistic design compels the model to rely solely on the provided schema and tool definitions to construct a valid reasoning plan.

The exact template for the prompt provided to the agent is shown below. Placeholders like `{graph_schema}` are dynamically populated at runtime.

```
You are a helpful assistant helping with Neo4j graph
database in a controlled environment. At your
disposal, you have a variety of tools, each
specialized in performing a distinct type of task.
For successful task completion, based on the
schema representation of the database, consider
the task at hand and determine which tool or
set of tools is best suited based on its
capabilities and the nature of the query. Each one of
the tools is equivalent to a cypher query. You can call
the tools to query the graph database and extract the
necessary information, but you cannot write a query
yourself. Please note that in order to get the right
answer, you might need to traverse the entire graph
database.

This is the graph schema representation
of the database:
<graph_schema>
{graph_schema}
</graph_schema>

System time:
<system_time>
{system_time}
</system_time>

<guidelines>
- Think step by step.
- If property values in the graph schema end with
  '...', it means the list is not exhaustive and you
  should obtain the full list from the graph database
  if needed.
- Use the tools to query the graph database and
  extract the necessary information.
- Remember that unless otherwise specified the tools
  are DETERMINISTIC, which means calling them with the
  same arguments again will return the same result and
  should be avoided.
- Provide the answer in the correct format that is
  requested in the question.
- If the user query is not answered by the tools, ask
  for additional information.
- Continue calling tools until you have all the
  necessary information needed to answer the user
  query. When you have the final answer, STOP CALLING
  ANY TOOLS.
</guidelines>
```

B Agent Toolset Descriptions

The agent is equipped with a minimal, orthogonal set of four tools to interact with the graph database. Three of these tools (`get_node_by_property`, `get_all_nearest_neighbors`, `get_unique_property_values`) are deterministic functions that map directly to Cypher queries. The fourth tool, `think`, is a special non-deterministic tool that allows the agent to record its reasoning process.

The exact docstrings provided to the agent for each tool are detailed below.

B.1 `get_node_by_property`

```
get_node_by_property(label, property_name,
property_value):
"""Retrieve a specific node from the graph
database by matching a property value.
  This tool searches for nodes with a specific
label that have a particular property set to
a given value. It's the primary way to find
specific entities in the graph when you know their
identifying property (like name, ID, or other
unique attribute).
  Use this when you need to:

   • Find a specific person, organization, drug,
     or other entity by name

   • Locate nodes with specific IDs or codes

   • Search for entities with particular
     attributes

  Args:
  label (str): The node label/type (e.g.,
"Person", "Drug", "Company").
     Must match exactly with labels in the graph
schema.
```

```
  property_name (str): The property to search by (e.g., "name", "id", "code").
      Must be a valid property for the specified label.
  property_value: The exact value to match. Can be string, number, or other types
      depending on the property. Must match exactly (case-sensitive for strings).
   Returns:
  list: List of matching nodes with all their properties. Each node is a dictionary
      containing all property key-value pairs for that node.
   Example:
  get_node_by_property("Person", "name", "John Smith")
  Returns: [{"name": "John Smith", "age": 30, "id": "person_123"}]
"""
```

B.2 get_all_nearest_neighbors

```
get_all_nearest_neighbors(label, property_name, property_value):
"""Get all directly connected neighbors of a specific node in the graph database.
   This tool finds a node by its property value and returns ALL nodes that are directly connected to it through any type of relationship. This is useful for exploring the immediate neighborhood of a node and understanding its direct connections.
   Use this when you need to:

    • Explore what entities are directly connected to a specific node

    • Find all immediate relationships of a person, organization, or other entity

    • Discover the local neighborhood around a node

    • Get a comprehensive view of direct connections before drilling down

   Args:
  label (str): The label/type of the central node (e.g., "Person", "Drug", "Company").
      Must match exactly with labels in the graph schema.
  property_name (str): The property to identify the central node (e.g., "name", "id").
      Must be a valid property for the specified label.
  property_value: The exact value to match for finding the central node.
      Must match exactly (case-sensitive for strings).
   Returns:
  list: List of all neighboring nodes with their properties and relationship information.
      Each result includes both the neighbor node data and the relationship that
      connects it to the central node.
   Example:
  get_all_nearest_neighbors("Person", "name", "John Smith")
  Returns all people, organizations, locations, etc. directly connected to John Smith
"""
```

B.3 get_unique_property_values

```
get_unique_property_values(property_name, entity_name, entity_type):
"""Retrieve all unique values for a specific property across all nodes or relationships of a given type.
   This tool is essential for data exploration and understanding what values exist in the database. It helps you discover available options, validate data, and understand the scope of information available for a particular entity type.
   Use this when you need to:

    • Explore what values are available for a property (e.g., all company names, or all relationship weights)

    • Validate if a specific value exists before searching

    • Get a complete list of options for categorical properties

    • Understand the data distribution and available entities

    • Find all possible values to choose from when building queries

   Args:
  property_name (str): The name of the property to get values for (e.g., "name",
      "category", "status"). Must be a valid property in the schema
      for the specified entity.
  entity_name (str): The node label or relationship type to examine (e.g., "Person", "Drug",
      "Company", "INTERACTS_WITH"). Must match exactly with labels/types in
      the graph schema.
  entity_type (str): The type of the entity to examine, which can be 'node' or 'relationship'.
   Returns:
  list: A list of dictionaries, each containing a unique value for the specified
      property. The structure is [{"values": value1}, {"values": value2}, ...].
   Example for a node:
  get_unique_property_values("name", "Company", "Node")
  Returns: [{"values": "Pfizer"}, {"values": "Johnson & Johnson"}, {"values": "Merck"}]
   Example for a relationship:
  get_unique_property_values("year", "MET_IN", "Relationship")
  Returns: [{"values": "2020"}, {"values": "2021"}]
"""
```

B.4 think

```
think(thought):
"""Record and process reasoning steps during graph traversal and query planning.
  This tool allows you to document your thought process, reasoning steps, and intermediate conclusions while working through complex graph queries. It's particularly valuable for multi-step problems where you need to plan your approach, track progress, or explain your reasoning.
  Use this when you need to:

  • Break down complex queries into logical steps

  • Document your reasoning for choosing specific
    tools or approaches

  • Summarize findings from previous tool calls
    before proceeding

  • Explain why you're taking a particular path
    through the graph

  • Keep track of progress in multi-step graph
    traversals

  • Clarify your understanding of the problem
    before answering

  Args:
  thought (str): Your reasoning, observation, or plan. Can include analysis of
      previous results, next steps to take, or explanations of your
      approach to solving the user's query.
  Returns:
  str: The same thought string you provided, allowing you to record and
      reference your reasoning process.

  Example:
  think("I found John Smith in the database. Now I need to find his company
      affiliations by looking at his neighbors, then find other employees
      of those companies.")
"""
```

C Question Templates and Categories

Our benchmark is composed of 12 distinct query templates, designed to evaluate a range of reasoning skills from simple retrieval to complex logical composition. The details for each template are provided in Table 4. For each template, we define the natural language instruction given to the agent, the ground-truth Cypher query that corresponds to the task, and the expected JSON output schema. Placeholders like `{source_label}` are populated dynamically from the specific random graph being used for the test run.

C.1 Performance by Question Category

Table 5 provides a granular breakdown of model performance across twelve distinct query types. This analysis reveals specific strengths and weaknesses tied to query complexity, tool utilization, and reasoning capabilities.

The models demonstrated the highest proficiency on single-step, direct retrieval tasks. Node by Property queries, which can typically be resolved with a single get_node_by_property tool call, were answered correctly 85 times, making it the most successfully handled category. Similarly, models performed well on Path from Specific Node (65 correct answers), likely because the maximum path length was constrained to three, keeping the search space manageable. The strong performance on Negation on Rel Property (55 correct answers) is notable and may be attributed to the detailed and explicit nature of the question templates for this category, which aids both in-context reasoning and tool-based query formulation.

Conversely, all of the models struggled with tasks requiring aggregation or some complex multi-step reasoning questions. Aggregation-based queries like Node Count (1 correct answer) and Relationship Count (2 correct answers) proved exceptionally difficult, even for tool-equipped models. This suggests a fundamental weakness in synthesizing information from multiple tool calls or serialized data points into a final aggregate value.

The most challenging categories were those requiring advanced logical composition and stateful exploration. Compositional Intersection (5 correct answers) and Negation with Connection (3 correct answers) highlight the difficulty models face in applying multiple logical constraints (AND/NOT) simultaneously. The complete failure on Variable Hop Path queries (0 correct answers) is particularly telling; these queries contain lengthy ground truth answers which appear to be beyond the models' ability to construct or reason over, indicating a critical limitation in their capacity for complex query planning and information aggregation.

Table 4: Detailed Breakdown of the Query Templates

Category	Question Class	Instruction Template	Output Schema	Ground Truth (Cypher)
Retrieval & Aggregation	Node Count	Count the number of "{source_label}" nodes that are connected to any "{target_label}" node. Return ONLY the output with the count in JSON format: {{output_schema}}.	`[{"count": "number"}]`	`MATCH (a:{source_label}) ->(b:{target_label}) RETURN count(DISTINCT a) AS count`
	Relationship Count	How many relationships of type "{rel_type_name}" exist? Return ONLY the output with the count in JSON format: {{output_schema}}.	`[{"count": "number"}]`	`MATCH ()-[r:{rel_type_name}]->() RETURN count(DISTINCT r) AS count`
	Node with Most Relationships	Which "{source_node_label}" node has the most outgoing "{rel_type_name}" relationships? Return ONLY ONE answer in JSON format as per the schema: {{output_schema}}.	`[{"node_key": "string", "rel_count": "number"}]`	`MATCH (n:{source_node_label}) -[r:{rel_type_name}]->() WITH n, count(r) AS rel_count RETURN n.key AS node_key, rel_count ORDER BY rel_count DESC LIMIT 1`
	Node by Property	Find all "{node_label}" nodes where "{prop_name}" is "{prop_value}". Return results in JSON format according to the schema: {{output_schema}}.	`[{"node_key": "string"}]`	`MATCH (n:{node_label} { {prop_name}: {query_prop_value} }) RETURN DISTINCT n.key AS node_key`
	Relationship by Property	Find all "{rel_type_name}" relationships where "{prop_name}" is "{prop_value}". Return results in JSON format based on the schema: {{output_schema}}.	`[{"source_key": "string", "target_key": "string", ...}]`	`MATCH (s)-[r:{rel_type_name} { {prop_name}: {query_prop_value} }]->(t) RETURN s.key as source_key, t.key as target_key, {return_clause}`
Path & Relational Traversal	Path Finding	Find all paths from "{source_label}" to "{target_label}" through "{middle_label}". Return results in JSON format as per schema: {{output_schema}}.	`[{"source_node_key": "string", "target_node_key": "string"}]`	`MATCH (a:{source_label})-> (b:{middle_label})-> (c:{target_label}) RETURN a.key AS source_node_key, c.key AS target_node_key`
	Variable Hop Path	Find all paths where a "{source_label}" node reaches a "{target_label}" node in 1 to {n} steps, then takes one more step to any other node. Return the keys of the source and target nodes in JSON format as per schema: {{output_schema}}.	`[{"source_node_key": "string", "target_node_key": "string"}]`	`MATCH (a:{source_label}) -[*1..{n}]-> (b:{target_label}) ->() RETURN DISTINCT a.key AS source_node_key, b.key AS target_node_key`
	Path from Specific Node	Find all paths of 1 to {n} steps from the node with key "{source_key}" to any node of type "{target_label}". Return the keys of the target nodes found in JSON format: {{output_schema}}.	`[{"target_node_key": "string"}]`	`MATCH (a:{source_label} {key: '{source_key}'}) -[*1..{n}]->(b:{target_label}) RETURN DISTINCT b.key AS target_node_key`
	Remote Node Property	From a "{source_label}" node with key "{source_key}" find a "{target_label}" node that is not a direct neighbor but is reachable in 2 or more hops, and return its "{prop_name}". ANY valid node's property will be accepted. Return ONLY ONE answer in JSON format: {{output_schema}}.	`[{"value": "{prop_type}"}]`	`MATCH (a:{source_label} {key: '{source_key}'}) -[*2..{self.max_hops}] ->(b:{target_label}) WHERE NOT (a)->(b) RETURN DISTINCT b.{prop_name} as value`

Table 4 – continued from previous page

Category	Question Class	Instruction Template	Output Schema	Ground Truth (Cypher)
Complex Logical Composition	Compositional Intersection	Find all nodes of type "{source_label}" that have a relationship to at least one "{target1_label}" node AND at least one "{target2_label}" node. Return the keys of these "{source_label}" nodes in JSON in this format: {{output_schema}}.	`[{"node_key": "string"}]`	`MATCH (a:{source_label}) WHERE EXISTS((a)->(:{target1_label})) AND EXISTS((a)->(:{target2_label})) RETURN DISTINCT a.key AS node_key`
	Negation with Connection	Find all nodes of type "{source_label}" that are connected to at least one "{positive_target_label}" node AND are not connected to any "{negative_target_label}" node. Return their keys in JSON in this format: {{output_schema}}.	`[{"node_key": "string"}]`	`MATCH (a:{source_label}) WHERE EXISTS((a)->(:{positive_target_label})) AND NOT EXISTS((a)->(:{negative_target_label})) RETURN DISTINCT a.key AS node_key`
	Negation on Rel Property	Find all "{source_label}" nodes where "{source_prop_name}" is "{source_prop_value}". From those, find the ones connected to a "{target_label}" node by a "{rel_type_name}" relationship where the relationship's "{prop_name}" is not "{val2}". Return the keys of the source nodes in JSON in this format: {{output_schema}}.	`[{"node_key": "string"}]`	`MATCH (a:{source_label} { {source_prop_name}: {query_source_prop_value} })-[r:{rel_type_name}]->(b:{target_label}) WHERE r.{prop_name} <> {query_rel_prop_value} RETURN DISTINCT a.key AS node_key`

Table 5: The number of correct answers by question category across all model configurations. T refers to tool use and NT refers to the tool-free alternative for the same model.

Category	Total	gpt-4o-mini		gpt-4o		gpt-4.1-nano		gpt-4.1-mini		gpt-4.1		o3-mini	o4-mini
		T	NT	T	NT	T	NT	T	NT	T	NT	NT	NT
Node Count	1	0	0	0	0	0	1	0	0	0	0	0	0
Relationship Count	2	0	0	0	0	0	1	0	1	0	0	0	0
Node with Most Relationships	26	3	0	3	0	0	1	3	2	3	2	1	8
Node by Property	85	10	0	10	6	7	0	10	5	10	7	10	10
Relationship by Property	9	0	0	0	0	0	0	1	0	0	0	0	8
Path Finding	7	0	0	0	0	0	0	0	0	1	1	0	5
Variable Hop Path	0	0	0	0	0	0	0	0	0	0	0	0	0
Path from Specific Node	65	2	4	8	6	2	3	8	4	7	7	4	10
Remote Node Property	30	1	1	4	3	1	0	4	3	4	3	2	4
Compositional Intersection	5	1	0	0	0	0	0	0	0	0	0	1	3
Negation with Connection	3	0	0	0	0	0	0	0	2	0	0	0	1
Negation on Rel Property	55	1	1	9	3	1	1	6	3	10	1	9	10