

Multi-Agent GraphRAG: A Text-to-Cypher Framework for Labeled Property Graphs

Anton Gusarov^{1, 2}, Anastasia Volkova², Valentin Khrulkov¹, Andrey Kuznetsov^{1, 2, 3},
Evgenii Maslov^{1, 2}, Ivan Oseledets^{1, 2}

¹AIRI, ²Skoltech, ³Innopolis University
Corresponding author: gusarov@airi.net

Preprint

Abstract

While Retrieval-Augmented Generation (RAG) methods commonly draw information from unstructured documents, the emerging paradigm of GraphRAG aims to leverage structured data such as knowledge graphs. Most existing GraphRAG efforts focus on Resource Description Framework (RDF) knowledge graphs, relying on triple representations and SPARQL queries. However, the potential of Cypher and Labeled Property Graph (LPG) databases to serve as scalable and effective reasoning engines within GraphRAG pipelines remains underexplored in current research literature. To fill this gap, we propose Multi-Agent GraphRAG, a modular LLM agentic system for text-to-Cypher query generation serving as a natural language interface to LPG-based graph data. Our proof-of-concept system features an LLM-based workflow for automated Cypher queries generation and execution, using Memgraph as the graph database backend. Iterative content-aware correction and normalization, reinforced by an aggregated feedback loop, ensures both semantic and syntactic refinement of generated queries. We evaluate our system on the CypherBench graph dataset covering several general domains with diverse types of queries. In addition, we demonstrate performance of the proposed workflow on a property graph derived from the IFC (Industry Foundation Classes) data, representing a digital twin of a building. This highlights how such an approach can bridge AI with real-world applications at scale, enabling industrial digital automation use cases.

Introduction

Graph databases are NoSQL systems designed to manage graph-structured data, typically based on the property graph model (Angles 2018). They are increasingly adopted across diverse domains from bioscience (Walsh, Mohamed, and Nováček 2020) to infrastructure (Donkers, Yang, and Baken 2020) as a powerful abstraction for representing interlinked entities with rich semantics, both in knowledge graphs and in specialized domain-specific databases. In this work, we explore how large language models (LLMs) can serve as natural language interface to complex technical data encoded in property graphs.

Retrieval-Augmented Generation (RAG) has emerged as a prominent strategy for grounding LLM outputs in structured or semi-structured sources. While most RAG approaches rely on unstructured documents or RDF-based

knowledge graphs, Labeled Property Graphs (LPGs) provide a more expressive alternative supporting rich attributes on both nodes and edges, flexible schemas, and native compatibility with declarative graph query languages such as Cypher.

However, applying RAG methods to LPGs introduces distinct challenges. Compared to RDF graphs with fixed ontologies and triple-based (subject-predicate-object) structures, LPGs exhibit greater schema variability and require support for multi-hop traversal, typed relationships, and attribute-level reasoning. These issues are amplified when adapting RAG pipelines for structured querying tasks, where the system must retrieve information by constructing correct and executable graph queries. Recent efforts have shown promise in structured generation, but primarily focus on tabular or RDF formats. In contrast, LLM-based querying over full-schema LPGs remains underexplored.

We address the gap in enabling natural language interfaces for Cypher-based question answering over property graphs, with a particular focus on high-impact domains such as digital construction. Our approach centers on a multi-agent GraphRAG workflow that interprets user queries, interacts with a graph database, and iteratively generates schema-compliant Cypher queries. As noted by (Han et al. 2025), GraphRAG pipelines must manage diverse, interdependent, and domain-specific information with non-transferable semantics. Building on this insight, we emphasize explicit query entity verification, LLM-driven observation, and database-grounded feedback to support robust iterative query refinement. In this paper we introduce *Multi-Agent GraphRAG*, a system for natural language querying over property graphs. Our architecture includes query generation, schema and query entity identifiers verification, execution, and feedback-driven refinement. We demonstrate its effectiveness on both benchmark (CypherBench (Feng, Papicchio, and Rahman 2025)) and domain-specific datasets, providing insights into its performance and limitations.

Labeled Property Graphs

A property graph is a labeled directed multi-graph defined as a tuple (Angles et al. 2024):

$$\mathcal{G} = (N, E, \rho, \lambda, \sigma)$$

where:

- N is a finite set of nodes (vertices),
- E is a finite set of edges (relationships), disjoint from N ,
- $\rho : E \rightarrow (N \times N)$ is a total function that assigns to each edge its source and target nodes,
- $\lambda : (N \cup E) \rightarrow \mathcal{P}_+(L)$ is a partial function assigning each node or edge a finite non-empty set of labels from an infinite label set L ,
- $\sigma : (N \cup E) \times P \rightarrow \mathcal{P}_+(V)$ is a partial function assigning each property key from the infinite set P a finite non-empty set of values from V .

Here, $\mathcal{P}_+(X)$ denotes the set of all finite non-empty subsets of a set X .

A *path* π in a property graph $\mathcal{G} = (N, E, \rho, \lambda, \sigma)$ is a finite alternating sequence of nodes and edges:

$$\pi = (n_1, e_1, n_2, e_2, \dots, e_k, n_{k+1})$$

such that $k \geq 0$, $n_i \in N$, $e_i \in E$, and for all $1 \leq i \leq k$, $\rho(e_i) = (n_i, n_{i+1})$.

The length of π is k , the number of edge identifiers. A path of length zero is simply a single node. The path label $\lambda(\pi)$ is the concatenation of the edge labels: $\lambda(e_1) \dots \lambda(e_k)$.

Related Work

While the Cypher language (Francis et al. 2018) has become the de-facto standard for querying *property graphs* in graph database systems like Neo4j and has significantly influenced the ISO/IEC 39075:2024 GQL standard (ISO/IEC 2024), its application to LLM-backed graph reasoning and structured knowledge extraction remains limited. In contrast, the Resource Description Framework (RDF) and the SPARQL query language have seen extensive adoption in semantic reasoning tasks and integration with large language models.

The existing literature on property graphs tends to emphasize algorithmic and model-level approaches over query-level execution and semantics.

GraphRAG. Retrieval augmented generation on graphs (GraphRAG) is a novel paradigm that extends conventional retrieval-augmented generation (RAG) by incorporating existing or generating on-the-go graph-structured data as a retrieval source for more accurate, interpretable, and multi-hop reasoning. Although research on GraphRAG has expanded, it remains fragmented with a focus on knowledge and document graphs limiting application in other domains (Han et al. 2025). According to (Peng et al. 2024) GraphRAG can be split into three stages: graph-based indexing, graph-guided retrieval, and graph-enhanced generation. Together, they enable structured retrieval and response synthesis by transforming graph data into formats optimized for LLM-based generation. GraphRAG proposed in (Edge et al. 2025) combines RAG and query-focused summarization by using an LLM to build an entity-based graph index and generate community summaries, enabling scalable question answering over large, private text corpora.

Knowledge Base Question Answering (KBQA). KBQA represents a task of generating precise answers to natural language queries by reasoning over structured knowledge

bases. KBQA approaches typically fall into two main categories (Luo et al. 2024): (i) semantic parsing methods, which convert natural language queries into executable logical forms over the knowledge base, and (ii) information retrieval methods, which retrieve relevant subgraphs or facts to inform the answer generation process (Jiang et al. 2024, 2023; Peng et al. 2024). The closely related concept of GraphRAG extends IR-based KBQA by leveraging graph-structured retrieval mechanisms, effectively treating IR-based KBQA as a special case within a broader framework for structured information access (Han et al. 2025).

Text-to-SQL Semantic Parsing. Natural language interfaces (NLI) to relational databases continue to attract vast attention in applied AI research. The sketch-based approach is one of the earliest methods of structured synthesis in general and of code in particular (Solar-Lezama 2009; Li et al. 2023b) that maps user queries to SQL by first predicting a high-level SQL pattern and then performing slot filling to generate the complete query (Xu, Liu, and Song 2017; Yu et al. 2018a).

Seq2SQL (Zhong, Xiong, and Socher 2017) introduces a sequence-to-sequence model with reinforcement learning to generate SQL queries from natural language, and belongs to the seq2seq (sequence-to-sequence) family of text-to-SQL methods (Li et al. 2023a).

Tree- or structure-based methods include bottom-up semantic parsers that construct SQL queries from semantically meaningful subtrees. A notable example is RAT-SQL (Wang et al. 2021) which introduces a relation-aware transformer that models the question and schema as a relational graph, enabling effective schema linking and structured representation learning for text-to-SQL parsing. SmBoP (Rubin and Berant 2021) employs a chart-based bottom-up decoder to generate SQL queries as compositional trees, coupled with the RAT-SQL encoder to jointly encode the user utterance and database schema.

SyntaxSQLNet (Yu et al. 2018b) introduces a syntax tree-based decoder that leverages SQL grammar to generate complex SQL queries, enabling generalization to unseen schemas and compositional structures. It belongs to the family of grammar-based or syntax-constrained decoding methods, which guide query generation using formal production rules to ensure syntactic correctness.

LLM Multi-Agent Text-to-SQL. Recent studies adopt an agent-based approach to Text-to-SQL generation leveraging modular design to split the task across multiple specialized agents. Unlike monolithic models, this setup promotes task-specific delegation and inter-agent collaboration for scalability and coordination. One example is CHASE-SQL (Pourreza et al. 2024) which harnesses multiple LLMs to generate diverse SQL candidates by decomposing complex queries, simulating execution plans via chain-of-thought reasoning, and providing instance-specific few-shot examples, with a dedicated selection agent ranking outputs through pairwise comparison using a fine-tuned LLM. Mac-SQL (Wang et al. 2025) framework features a central decomposer agent for few-shot chain-of-thought Text-to-SQL generation, supported by expandable auxiliary agents that dynamically as-

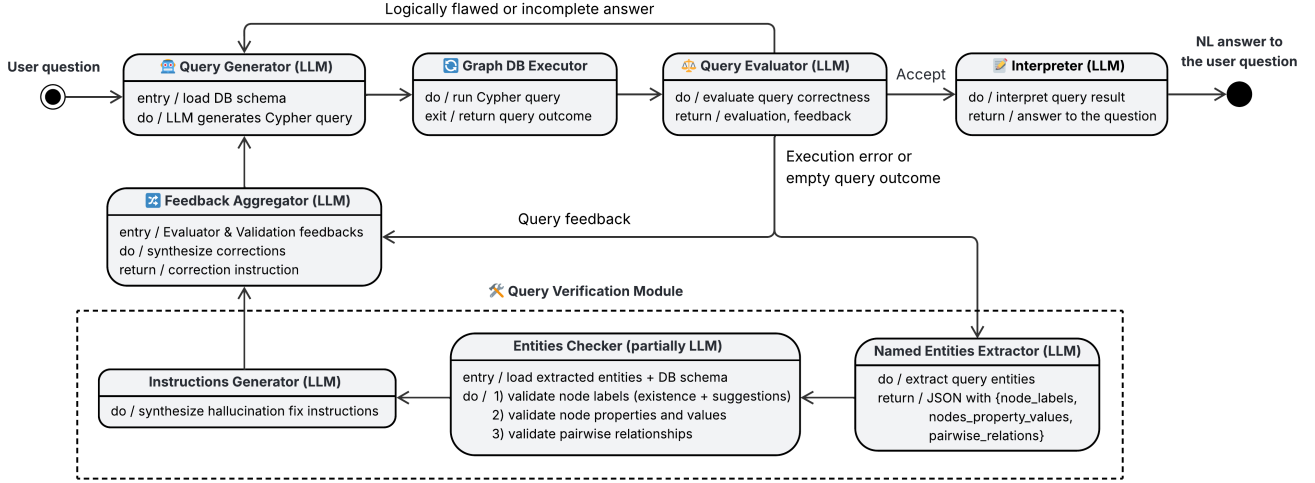


Figure 1: State diagram outlining the proposed Multi-Agent GraphRAG workflow for Cypher-based information retrieval on property graphs.

sist with sub-database retrieval and SQL refinement. Alpha-SQL (Li et al. 2025) tackles the challenges of zero-shot Text-to-SQL by combining Monte Carlo Tree Search (MCTS) with an LLM-based action model and self-supervised reward.

Agent-Based Graph Reasoning. Several recent papers on iterative agent-based methods aimed at graph data.

The Graph Chain-of-Thought solution (Jin et al. 2024) tackles the LLM graph reasoning problem with three-step iterations: reasoning, interaction, and execution.

Based on Graph-CoT ideas, (Gao et al. 2025) proposes a multi-agent GraphRAG approach addressing QA tasks with cooperative agentic KG exploration and extraction strategies search supported by a multi-perspective self-reflection module that refines reasoning strategy.

Text-to-Cypher. (Hornsteiner et al. 2024) develop a modular natural language interface for Neo4j using GPT-4 Turbo to perform Cypher query generation, database selection, and error correction, employing the design science research methodology and reporting high accuracy in few-shot scenarios for practical Text-to-Cypher tasks on CLEVR graph dataset (Mack and Jefferson 2018), representing a transport transit network.

(Chatterjee and Dethlefs 2022) present an automated QA system for wind turbine maintenance that uses a semantic parser to convert natural language queries into Cypher for interactive decision-making support, and release a domain-specific dataset of question-Cypher pairs for this purpose.

Quite a few question-to-Cypher datasets are publicly available. SpCQL (Guo et al. 2022) introduces the first large-scale Text-to-Cypher (Text-to-CQL) semantic parsing dataset containing 10,000 natural language and Cypher query pairs over a Neo4j graph, highlighting the unique challenges of Cypher compared to SQL and exposing the limitations of existing Text-to-SQL models when applied

to graph databases. However, this is based on the Chinese language OwnThink knowledge graph and not available for direct download. (Feng, Papicchio, and Rahman 2025) present CypherBench, a benchmark for evaluating LLM-based question answering over full-schema property graphs using Cypher, built on Wikidata-derived datasets with diverse query types aligned to realistic graph schemas. (Tiwari et al. 2025) introduce Auto-Cypher, a fully automated LLM-supervised pipeline for generating and verifying synthetic Text2Cypher training data (SynthCypher) and introduces an adapted SPIDER-Cypher benchmark to address the lack of evaluation standards in this domain.

In comparison to prior work that primarily focus on static prompt-based Cypher generation or dataset construction for benchmarking, our approach introduces a modular multi-agent GraphRAG workflow that emphasizes iterative refinement, named entity verification, and aggregated semantic-syntactic feedback. Crucially, our system integrates runtime interaction with the graph database to detect and correct both hallucinations and logical errors. In contrast to (Hornsteiner et al. 2024), which targets Neo4j, we implement our workflow on Memgraph, demonstrating broader applicability to LPG-compatible backends where runtime efficiency is critical. Additionally, we test the system on industry-grade use cases, specifically IFC-based digital building representations, extending beyond general-purpose or synthetic datasets into the AEC (Architecture, Engineering, and Construction) domain.

Implementation

We propose an LLM workflow that adopts a modular agentic architecture with a feedback-driven refinement loop to overcome the limitations of standalone LLMs in text-to-Cypher generation by leveraging their reasoning capabilities for iterative improvement (Qu et al. 2024). Figure 1 provides an overview of the system components and data flow.

The implementation code along with the experimental results is publicly available at: <https://github.com/your-repo>.

Agent Roles and Responsibilities. The workflow is composed of seven cooperating agents and one graph database query executor module, each specializing in a distinct sub-task.

1. **Query Generator** formulates a Cypher query based on the user question in natural language. The query should be both syntactically correct and semantically aligned with the user intent. The generation is grounded on the provided context graph schema to the model, including node and pairwise relationship descriptions. In subsequent iterations, if the first-shot generation was not accepted, the agent takes aggregated feedback from the Evaluator and Verification agents to refine the previous revision of the Cypher query.
2. **Graph Database Executor** interfaces with the underlying graph database engine (Memgraph in our case) to execute the generated Cypher query and retrieve the outcome, which may consist of structured result data, an error message, or an empty result set. The latter is common when the Query Generator’s LLM incorrectly references attribute names, often due to hallucinations or insufficient data in the graph schema.
3. **Query Evaluator** is responsible for assessing the semantic and logical adequacy of the generated Cypher query relative to the user’s intent and the correctness of the query results. Functionally, it serves as an *LLM-based critic* (McAleese et al. 2024; Yang et al. 2025) within our pipeline. The evaluator is prompted to analyze three key aspects: (1) consistency between the user’s intended semantics and its natural language explanation, (2) alignment of the query logic with the user question, and (3) validity and informativeness of the returned results. Based on this assessment, it outputs both structured feedback and a discrete query grade from the set:
 - **Accept:** the query is error-free and the returned results fully and logically answer the user question.
 - **Incorrect:** the query executes without error and returns data, but is semantically misaligned, logically flawed, or incomplete.
 - **Error or Empty:** the query either fails to execute due to a runtime error or returns no results.The last case commonly arises from misidentified named entities, overly restrictive conditions, or invalid traversal paths in the graph often due to LLM hallucinations.
4. **Named Entity Extractor** is an LLM-based component that identifies elements within the query such as node labels, property-value pairs, and relationship types that are susceptible to hallucination. Its primary function is to decompose the query to enable subsequent verification of their existence in the underlying graph data.
5. **Verification Module** follows the Named Entity Extractor and for verifies the existence and correctness of the extracted schema elements against the actual graph data. This is done first programmatically via auxiliary Cypher

queries to the database. For any entity not found, indicating likely hallucination, the module initiates a two-step recovery process. First, it retrieves candidate replacements based on the normalized Levenshtein similarity ratio (Bachmann 2024). Second, it leverages an LLM to semantically rank all existing entities of the same type in the database, selecting the most contextually appropriate alternative.

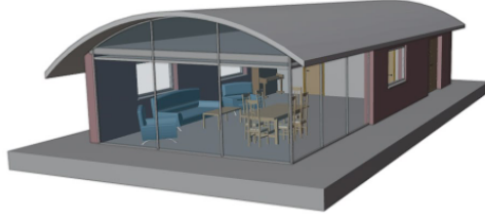
6. **Instructions Generator** synthesizes revision instructions based on the verification results targeting hallucinated or misnamed entities. It takes as input the structured feedback from the Verification Module, including which entities failed to match the data content. For each invalid component, it generates a correction proposal by combining: (i) edit-based suggestions derived from high-similarity alternatives, and (ii) semantically grounded recommendations ranked by the LLM. The output is a concise instruction to guide the Query Generator on how to revise the query.
7. **Feedback Aggregator** integrates the outputs of both the Query Evaluator and the Verification Module into a unified correction strategy. It consolidates signals such as semantic inconsistencies, execution errors, and naming mismatches to produce structured and prioritized feedback. This aggregated feedback is the basis for guiding the subsequent correction of the query, ensuring that both logical soundness and schema compliance are addressed.
8. Accepted queries are passed to the **Interpreter**, which finally generates a concise, domain-relevant natural language answer.

Each agent in our pipeline is guided by a system prompt tailored to its role. The complete prompt templates are provided in the public implementation: <https://github.com/your-repo>.

Self-Correction Loop. The iterative schema-aware correction and normalization is a critical mechanism for boosting query accuracy and semantic alignment. It operates over a maximum of four iterations, progressively improving the Cypher query by incorporating feedback from both semantic validation and data-level verification. At each step, the system analyzes execution outcomes, detects logical flaws or schema mismatches, and generates targeted correction instructions. The refinement loop is detailed in Algorithm 1.

Incorporating Graph Schema into LLM Prompts. The LLM is data-informed such that the graph database schema is explicitly incorporated into the system prompt of the Query Generator agent. We experimentally observed that generation quality significantly improves when the schema is presented in a format closely resembling actual Cypher query syntax, as this provides better structural grounding and improves token-level alignment with expected outputs.

For each node type, representative attribute–value examples are also provided to guide the generation of property-based query conditions. Detailed listings of the node and relationship schema used for the *fictional character* graph from the CypherBench dataset (Feng, Papicchio, and Rahman 2025) are included in Appendix A.



```

...
#42= IFCOWNERHISTORY(#39,#5,$,NOCHANGE,$,$,$,1418982422);
...
#177= IFCSPACE('3w0zWKm7n8SB1qbfwUzt0U',#42,'1 - Living room',$,$,#140,#172,'Living room',ELEMENT,SPACE,$);
#182= IFCPROPERTYSET('Reference',$,IFCIDENTIFIER('Living room 1 - Living room'),$);
#188= IFCPROPERTYSET('IsExternal',$,IFCBOOLEAN(F,$));
#189= IFCPROPERTYSET('1JkF2aGfH7699uBSGTW7Nn',#42,'Pset_SpaceCommon',$,($182,$188));
#198= IFCRELDEFINESBYPROPERTIES('1bmHBnuzj9MAnr4b0suYVn',#42,$,$,($177,$189));
#202= IFCQUANTITYAREA('NetFloorArea','area measured in geometry',$,$,1.9948250000001,$);
#205= IFCQUANTITYLENGTH('Height','length measured in geometry',$,$,2500,$);
#206= IFCQUANTITYLENGTH('GrossPerimeter','length measured in geometry',$,$,29925,$);
...

```

Figure 2: Single-storey *Sample House* IFC model (xBIM Team 2024), first utilized for GraphRAG-based information extraction in (Iranmanesh, Saadany, and Vakaj 2025). Left: 3D representation of the building. Right: fragment illustrating the cross-relations of IFC entities mapped into a labeled property graph, including spatial hierarchies, property sets, and quantitative attributes.

Algorithm 1: Cypher query refinement with semantic validation and named entities verification

Input: User question Q , nodes and relationships schema S

Output: Valid Cypher query C and NL answer A

```

1:  $C \leftarrow \text{GenerateQuery}(Q, S)$ 
2:  $R \leftarrow \text{ExecuteQuery}(C)$ 
3:  $E \leftarrow \text{EvaluateQuery}(Q, C, R)$ 
4:  $t \leftarrow 1$  // iteration counter
5: while  $E_{\text{status}} \neq \text{Accept}$  and  $t \leq 4$  do
6:   if  $E_{\text{status}} = \text{Illogic or Incomplete}$  then
7:      $C \leftarrow \text{GenerateQuery}(Q, S, E_{\text{feedback}})$ 
8:   else if  $E_{\text{status}} = \text{Error or Empty}$  then
9:      $E_{\text{ent}} \leftarrow \text{ExtractNamedEntities}(C)$ 
10:     $V \leftarrow \text{VerifyNamedEntities}(E_{\text{ent}}, \text{Graph})$ 
11:    // check existence and suggest replacements for:
12:    - Node labels
13:    - Node property values
14:    - Pairwise edge patterns
15:     $I \leftarrow \text{GenerateFixInstructions}(V)$ 
16:     $F \leftarrow \text{AggregateFeedback}(Q, E_{\text{feedback}}, I)$ 
17:     $C \leftarrow \text{GenerateQuery}(Q, S, F)$ 
18:   end if
19:    $t \leftarrow t + 1$ 
20:    $R \leftarrow \text{ExecuteQuery}(C)$ 
21:    $E \leftarrow \text{EvaluateQuery}(Q, C, R)$ 
22: end while
23:  $A \leftarrow \text{InterpretResult}(Q, R)$ 
24: return  $A, C$ 

```

Implementation Details. The proposed Multi-Agent GraphRAG system is implemented in Python 3.12 using the LangGraph framework for agent orchestration and asynchronous state management. The underlying graph database is Memgraph (Memgraph Ltd. 2025), an in-memory, with C++ backed, open-sourced graph engine optimized for low-latency query execution. LLM agents are integrated via an API with OpenAI-compatible interface.

Experiments

Experimental Setup

Datasets. We evaluate our Multi-Agent RAG system on a subset of the CypherBench benchmark (Feng, Papicchio, and Rahman 2025), a recently introduced dataset for evaluating question answering over property graphs using Cypher. CypherBench includes realistic, large-scale knowledge graphs extracted from Wikidata, each accompanied by a set of natural language questions aligned with the underlying schema and publicly available graph. From this suite, we selected five diverse graphs: *art*, *flight accident*, *company*, *geography*, and *fictional character* to assess the generalization ability of our agentic pipeline across varied domains.

For each graph, we randomly sampled 150 question-answer pairs to ensure a balanced evaluation across domains. These graphs exhibit heterogeneous schemas and for the *geography* graph, we additionally include our re-processed schema in Appendix A to illustrate how structured schema information was incorporated into the Query Generator context. To the best of our knowledge, CypherBench is the only large-scale text-to-Cypher benchmark that offers fully public access to both multi-domain property graphs and text-to-Cypher question-query pairs at scale with 7.8 million entities and 11 subject domains.

To evaluate the feasibility of our multi-agent GraphRAG pipeline in real-world engineering scenarios, we additionally used a graph derived from the Industry Foundation Classes (IFC) standard data widely adopted for building information modeling (BIM) (Vanlande, Nicolle, and Cruz 2008) in architecture, engineering, and construction (AEC). IFC data, due to its object-oriented structure and typed relationships, naturally maps to a labeled property graph model, making it well suited for Cypher-based reasoning for information inquiry.

We use the publicly available single-storey house IFC model (xBIM Team 2024) we refer to as the *Sample House* and its corresponding graph representation from the dataset introduced in (Iranmanesh, Saadany, and Vakaj 2025), which includes 10 manually curated natural language questions of

Dataset	Gemini 2.5 Pro		GPT-4o (2024-11-20)		Qwen3 Coder		GigaChat 2 MAX	
	Single	Agentic	Single	Agentic	Single	Agentic	Single	Agentic
art	55.70%	63.33%	55.03%	56.85%	32.00%	51.33%	30.67%	40.00%
flight accident	82.67%	92.00%	78.00%	86.67%	74.00%	76.00%	67.33%	75.33%
company	59.33%	68.46%	46.00%	48.00%	30.67%	31.33%	25.50%	38.26%
geography	68.00%	76.35%	54.00%	63.09%	47.33%	56.00%	45.33%	54.67%
fictional character	69.33%	86.01%	47.33%	59.68%	44.67%	52.38%	37.33%	47.95%
Average	67.00%	77.23%	56.07%	62.86%	45.73%	53.40%	41.23%	51.24%

Table 1: Accuracy comparison between linear-pass LLM baseline and our multi-agent text-to-Cypher generation system across CypherBench domains and models.

varying structural complexity. This evaluation setup allows us to demonstrate the applicability of our method beyond open-domain knowledge graphs and into structured engineering data contexts. A visualization of the sample building and IFC data structure is shown in Figure 2.

Baseline and Models. We evaluate our proposed multi-agent Cypher generation pipeline against a baseline implemented using four state-of-the-art LLM backbones: *Gemini 2.5 Pro*, *GPT-4o (2024-11-20)*, *GigaChat 2 MAX*, and *Qwen3 Coder*. Each model is tested under two configurations: (i) **“Single”**, a linear-pass baseline where only the Query Generator → Executor → Interpreter sequence performs the full answer generation task without iterative feedback but (up to four attempts also allowed); and (ii) **“Agentic”**, our Multi-Agent GraphRAG setup, in which the same model drives the proposed multi-agent workflow with distinct roles for evaluation, verification, and iterative Cypher query refinement.

Evaluation Metrics. We report *accuracy* as the percentage of correctly answered questions in natural language form. Answer correctness is assessed using a reference-based evaluation procedure, where a dedicated LLM compares the generated natural language answer against the given in a form of JSON database outcome ground truth. Following the LLM-as-a-judge framework (Tiwari et al. 2025), we employ *GigaChat 2 MAX* as the evaluation model to judge semantic equivalence between the answers. This automated evaluation approach allows consistent and scalable evaluation across all tested models and domains. The prompt of a judge LLM apart from instructions contains few-shot examples and available in the project’s repository.

Experimental Results

CypherBench dataset. Table 1 presents the accuracy results following LLM-as-a-judge metric across five selected CypherBench dataset domains for four foundation models in both single-pass (without iterative refinement) and multi-agent settings. The final row reports the average performance across all evaluated datasets. The pipeline was executed once per dataset graph to obtain accuracy results, allowing up to four refinement attempts per query.

According to these results, Multi-Agent GraphRAG pipeline consistently outperforms the linear-pass LLM base-

line across all models and domains in the experiment. On average, the proposed agentic workflow yields noticeable improvements: on average +10.23% for *Gemini 2.5 Pro*, +6.79% for *GPT-4o*, +7.67% for *Qwen3 Coder* and +10.01% for *GigaChat 2 MAX*. The results indicate that incorporating iterative refinement, verification, and semantics-syntax feedback aggregation enhances structured query generation capabilities in LLM-based information retrieval systems over the property graphs.

Sample IFC data. Table in Appendix C presents the results of our workflow using *Gemini 2.5 Pro* on the architectural *Sample House* IFC dataset, which contains ten ground-truth question-answer pairs. Following the evaluation protocol of (Iranmanesh, Saadany, and Vakaj 2025), we report all questions and answers in full. Their method and results are used as the baseline for comparison. Compared to (Iranmanesh, Saadany, and Vakaj 2025), our Multi-Agent GraphRAG system correctly answers the last three questions (previously unanswered or answered partially) and shows the ability to express uncertainty (e.g., Question 2) and grounding responses on the graph database contents (e.g., Questions 7 and 8).

Discussion and limitations

In this section, we analyze the performance of the proposed workflow by tracing how its components process data, make decisions and refine Cypher query from iteration to iteration. Drawing on qualitative analysis of system traces (see example in Appendix B), we highlight factors contributing to successful query generation and identify remaining limitations. All experimental traces are publicly available together with the system’s code implementation.

The effectiveness of the Multi-Agent GraphRAG stems from several key design components. First of all, schema validation and query names normalization play a crucial role. Database-grounded feedback enables correction of structural errors such as misused relationship directions or types, while entity verification mitigates LLM hallucinations by enforcing consistency with actual graph database content. For example, when the system generates a query on employees and their managers, database-informed feedback pushes correct relationship traversal (e.g., from employee to manager via `Reports.to` relationship), while entity verification ensures that department names or employee iden-

tifiers match actual entries via auxiliary database queries. Together, they support multi-hop reasoning and allow the system to iteratively converge to schema-compliant and executable queries.

The workflow also benefits from reformulating Cypher’s strict comparisons into explicit value retrieval. Rather than relying on binary equality checks that may yield empty or ambiguous results, returning relevant property values for each entity allows it to infer the answer more transparently. For instance, when asked whether two characters share the same creator, the system first attempted a direct equality check (`c1.creator = c2.creator`), which returned nothing due to a mismatch. A subsequent reformulation `MATCH (c:Character) WHERE c.name IN ["Morbis", "the Living Vampire", "Giganto"] RETURN c.name, c.creator` explicitly retrieves each character’s creator, avoiding empty results and making the comparison observable in the output.

Despite these strengths, several limitations remain, which we outline below based on failure cases observed in the traces. One observed limitation is the difficulty in handling compositional queries involving disjunctions (e.g., “*Who are married to Cersei Lannister or have Cassana Baratheon as their mother?*”), which requires the union of two structurally distinct subqueries) and symmetric relationships (e.g. `(:Character)-[:hasSpouse]-(other:Character)`), which can match from either side and therefore complicates schema validation and query formulation) even with multi-step feedback. Addressing these cases may require explicit query planning or intermediate symbolic representations of query intent.

The Multi-Agent GraphRAG also struggles with multi-intent questions that require decomposing and aligning distinct subgoals such as e.g. listing children and counting their descendants (in CypherBench’s *fictional character*) leading to semantic conflation and misaligned answer structure. This highlights a limitation in handling compositional queries where sub-intents must be separated and resolved independently.

These findings suggest that the system’s success is tied to its ability to integrate database-aware verification, semantic-syntactic feedback, and iterative refinement. At the same time, addressing the remaining limitations, particularly for compositional and structurally complex Cypher queries, opens a way for future improvements in agentic query generation over PLG graphs.

Conclusion

We presented the Multi-Agent GraphRAG system for text-to-Cypher question answering over property graph databases. Our approach combines modular LLM-agentic components for query generation, query entities verification, execution, and feedback aggregation into an iterative refinement loop. Experimental results across CypherBench and IFC-derived datasets demonstrate that this design improves query accuracy and robustness compared to existing LLM baselines.

Unlike prior work focused primarily on Neo4j, our system targets Memgraph, expanding the landscape of LLM-based

structured querying. Furthermore, while previous systems often rely on static schemas, our refinement loop dynamically interacts with the database through auxiliary Cypher queries, enabling more adaptive and context-aware correction strategies.

These systems show promise as natural interfaces to complex domain-specific data. In our study, we demonstrated performance on IFC (Industry Foundation Classes) sample data – a widely adopted format for representing buildings in the AEC (Architecture, Engineering, and Construction) sector, highlighting the value of such pipelines for enabling simplified access to complex structured technical data.

Future work includes extending this approach to multi-turn dialogue scenarios, incorporating explicit subgoal planning for compositional queries, and developing larger domain-specific datasets, particularly for IFC-linked Cypher queries, to support the adoption of AI-driven solutions in digital construction and operation.

References

- Angles, R. 2018. The Property Graph Database Model. In *Alberto Mendelzon Workshop on Foundations of Data Management*.
- Angles, R.; Bonifati, A.; García, R.; and Vrgoč, D. 2024. Path-based Algebraic Foundations of Graph Query Languages. arXiv:2407.04823.
- Bachmann, M. 2024. RapidFuzz: Fuzzy String Matching for Python. <https://rapidfuzz.github.io/RapidFuzz/index.html>. Accessed: 2025-07-30.
- Chatterjee, J.; and Dethlefs, N. 2022. Automated Question-Answering for Interactive Decision Support in Operations and Maintenance of Wind Turbines. *IEEE Access*.
- Donkers, A.; Yang, D.; and Baken, N. 2020. Linked data for smart homes: comparing RDF and labeled property graphs. In *LDAC*.
- Edge, D.; Trinh, H.; Cheng, N.; Bradley, J.; Chao, A.; Mody, A.; Truitt, S.; Metropolitansky, D.; Ness, R. O.; and Larson, J. 2025. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130.
- Feng, Y.; Papicchio, S.; and Rahman, S. 2025. Cypher-Bench: Towards Precise Retrieval over Full-scale Modern Knowledge Graphs in the LLM Era. arXiv:2412.18702.
- Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; and Taylor, A. 2018. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, 1433–1445.
- Gao, J.; Zou, X.; Ai, Y.; Li, D.; Niu, Y.; Qi, B.; and Liu, J. 2025. Graph Counselor: Adaptive Graph Exploration via Multi-Agent Synergy to Enhance LLM Reasoning. arXiv:2506.03939.
- Guo, A.; Li, X.; Xiao, G.; Tan, Z.; and Zhao, X. 2022. Spcql: A semantic parsing dataset for converting natural language into cypher. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, 3973–3977.

- Han, H.; Wang, Y.; Shomer, H.; Guo, K.; Ding, J.; Lei, Y.; Halappanavar, M.; Rossi, R. A.; Mukherjee, S.; Tang, X.; He, Q.; Hua, Z.; Long, B.; Zhao, T.; Shah, N.; Javari, A.; Xia, Y.; and Tang, J. 2025. Retrieval-Augmented Generation with Graphs (GraphRAG). arXiv:2501.00309.
- Hornsteiner, M.; Kreussel, M.; Steindl, C.; Ebner, F.; Empl, P.; and Schöning, S. 2024. Real-time text-to-cypher query generation with large language models for graph databases. *Future Internet*, 16(12): 438.
- Iranmanesh, S.; Saadany, H.; and Vakaj, E. 2025. LLM-assisted Graph-RAG Information Extraction from IFC Data. arXiv:2504.16813.
- ISO/IEC. 2024. ISO/IEC 39075:2024 Information technology – Database languages – Graph Query Language (GQL). Technical Report ISO/IEC 39075:2024, ISO/IEC. Standard.
- Jiang, J.; Zhou, K.; Zhao, W. X.; Song, Y.; Zhu, C.; Zhu, H.; and Wen, J.-R. 2024. KG-Agent: An Efficient Autonomous Agent Framework for Complex Reasoning over Knowledge Graph. arXiv:2402.11163.
- Jiang, J.; Zhou, K.; Zhao, W. X.; and Wen, J.-R. 2023. UniKGQA: Unified Retrieval and Reasoning for Solving Multi-hop Question Answering Over Knowledge Graph. arXiv:2212.00959.
- Jin, B.; Xie, C.; Zhang, J.; Roy, K.; Zhang, Y.; Li, Z.; Li, R.; Tang, X.; Wang, S.; Meng, Y.; and Han, J. 2024. Graph Chain-of-Thought: Augmenting Large Language Models by Reasoning on Graphs. In Ku, L.-W.; Martins, A.; and Srikanth, V., eds., *The 62nd Annual Meeting of the Association for Computational Linguistics*, Proceedings of the Annual Meeting of the Association for Computational Linguistics, 163–184. Association for Computational Linguistics (ACL). Publisher Copyright: © 2024 Association for Computational Linguistics.; Findings of the 62nd Annual Meeting of the Association for Computational Linguistics, ACL 2024 ; Conference date: 11-08-2024 Through 16-08-2024.
- Li, B.; Zhang, J.; Fan, J.; Xu, Y.; Chen, C.; Tang, N.; and Luo, Y. 2025. Alpha-SQL: Zero-Shot Text-to-SQL using Monte Carlo Tree Search. arXiv:2502.17248.
- Li, H.; Zhang, J.; Li, C.; and Chen, H. 2023a. Resdsql: Decoupling schema linking and skeleton parsing for text-to-sql. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 13067–13075.
- Li, J.; Li, Y.; Li, G.; Jin, Z.; Hao, Y.; and Hu, X. 2023b. Skcoder: A sketch-based approach for automatic code generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2124–2135. IEEE.
- Luo, H.; E, H.; Tang, Z.; Peng, S.; Guo, Y.; Zhang, W.; Ma, C.; Dong, G.; Song, M.; Lin, W.; Zhu, Y.; and Luu, A. T. 2024. ChatKBQA: A Generate-then-Retrieve Framework for Knowledge Base Question Answering with Fine-tuned Large Language Models. In *Findings of the Association for Computational Linguistics ACL 2024*, 2039–2056. Association for Computational Linguistics.
- Mack, D.; and Jefferson, A. 2018. CLEVR graph: A dataset for graph question answering. <https://github.com/Octavian-ai/clevr-graph>. Accessed: 2025-07-29.
- McAleese, N.; Pokorny, R. M.; Uribe, J. F. C.; Nitishinskaya, E.; Trebacz, M.; and Leike, J. 2024. LLM Critics Help Catch LLM Bugs. arXiv:2407.00215.
- Memgraph Ltd. 2025. Memgraph Database. <https://memgraph.com>. Accessed: 2025-07-31.
- Peng, B.; Zhu, Y.; Liu, Y.; Bo, X.; Shi, H.; Hong, C.; Zhang, Y.; and Tang, S. 2024. Graph Retrieval-Augmented Generation: A Survey. arXiv:2408.08921.
- Pourreza, M.; Li, H.; Sun, R.; Chung, Y.; Talaei, S.; Kakkar, G. T.; Gan, Y.; Saberi, A.; Ozcan, F.; and Arik, S. O. 2024. CHASE-SQL: Multi-Path Reasoning and Preference Optimized Candidate Selection in Text-to-SQL. arXiv:2410.01943.
- Qu, Y.; Zhang, T.; Garg, N.; and Kumar, A. 2024. Recursive introspection: Teaching language model agents how to self-improve. *Advances in Neural Information Processing Systems*, 37: 55249–55285.
- Rubin, O.; and Berant, J. 2021. SmBoP: Semi-autoregressive Bottom-up Semantic Parsing. arXiv:2010.12412.
- Solar-Lezama, A. 2009. The sketching approach to program synthesis. In *Asian symposium on programming languages and systems*, 4–13. Springer.
- Tiwari, A.; Malay, S. K. R.; Yadav, V.; Hashemi, M.; and Madhusudhan, S. T. 2025. Auto-Cypher: Improving LLMs on Cypher generation via LLM-supervised generation-verification framework. arXiv:2412.12612.
- Vanlande, R.; Nicolle, C.; and Cruz, C. 2008. IFC and building lifecycle management. *Automation in construction*, 18(1): 70–78.
- Walsh, B.; Mohamed, S. K.; and Nováček, V. 2020. BioKG: A Knowledge Graph for Relational Learning On Biological Data. *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*.
- Wang, B.; Ren, C.; Yang, J.; Liang, X.; Bai, J.; Chai, L.; Yan, Z.; Zhang, Q.-W.; Yin, D.; Sun, X.; and Li, Z. 2025. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. arXiv:2312.11242.
- Wang, B.; Shin, R.; Liu, X.; Polozov, O.; and Richardson, M. 2021. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers. arXiv:1911.04942.
- xBIM Team. 2024. SampleHouse4.ifc: Example Building Model in IFC Format. <https://raw.githubusercontent.com/xBimTeam/XbimEssentials/refs/heads/master/Tests/TestFiles/SampleHouse4.ifc>. Accessed: 2025-07-31.
- Xu, X.; Liu, C.; and Song, D. 2017. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. arXiv:1711.04436.
- Yang, R.; Ye, F.; Li, J.; Yuan, S.; Zhang, Y.; Tu, Z.; Li, X.; and Yang, D. 2025. The Lighthouse of Language: Enhancing LLM Agents via Critique-Guided Improvement. arXiv:2503.16024.
- Yu, T.; Li, Z.; Zhang, Z.; Zhang, R.; and Radev, D. 2018a. TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation. arXiv:1804.09769.

Yu, T.; Yasunaga, M.; Yang, K.; Zhang, R.; Wang, D.; Li, Z.; and Radev, D. 2018b. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-DomainText-to-SQL Task. arXiv:1810.05237.

Zhong, V.; Xiong, C.; and Socher, R. 2017. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. arXiv:1709.00103.

Appendix A

The listings below provide example schemas used to guide the Query Generator agent during Cypher query generation. They include representative node and relationship definitions for the *fictional character* graph in the CypherBench dataset (Feng, Papicchio, and Rahman 2025), note that the schemas are formatted to closely resemble Cypher syntax.

Listing 1: Example node schema with properties and sampled values from the *fictional character* graph in CypherBench dataset

```
Each node type includes properties
hierarchy divided by '.' and sampled
examples of each property values.

Node Type: Character
Properties:
  .aliases: "Ibuki Suika"
  .birth_name: Thomas Merlyn
  .country_of_citizenship: "Denmark"
  .creator: Ake Holmberg
  .description: adventure time character
  .gender: trans woman
  .name: Thunderbolt (DC Comics)
  .occupation: "prophet, psychic"

Node Type: FictionalUniverse
Properties:
  .aliases: "SoD universe"
  .copyright_holder: Sony Group
  .creator: CD Projekt RED
  .description: self-contained narrative
                  universe of the SCP web-based
                  collaborative writing project
  .inception_year: 1999
  .name: The Black Hole universe

Node Type: Location
Properties:
  .aliases: "Jiangzhou Fu"
  .description: Klingon penal
                  colony in Star Trek
  .name: Erui

Node Type: Organization
Properties:
  .aliases: "Neo Team Rocket"
  .description: fictional tribe
                  in Far Cry 3
  .name: Wayne Enterprises
```

Listing 2: Example relationships schema from the *fictional character* graph in CypherBench

```
Type: basedIn
- (:Organization)-[:basedIn]->(:Location)

Type: bornIn
- (:Character)-[:bornIn]->(:Location)

Type: diedIn
- (:Character)-[:diedIn]->(:Location)

Type: fromUniverse
- (:Character)-[:fromUniverse]->
  (:FictionalUniverse)
- (:Organization)-[:fromUniverse]->
  (:FictionalUniverse)

Type: hasFather
- (:Character)-[:hasFather]->(:Character)

Type: hasMother
- (:Character)-[:hasMother]->(:Character)

Type: hasSpouse
- (:Character)-[:hasSpouse]->(:Character)

Type: hasStudent
- (:Character)-[:hasStudent]->(:Character)

Type: killedBy
- (:Character)-[:killedBy]->(:Character)

Type: memberOf
- (:Character)-[:memberOf]->
  (:Organization)
```

Appendix B

An example trace of the proposed Multi-Agent GraphRAG pipeline's question-answering process for a question from the *fictional character* graph in CypherBench (Feng, Papicchio, and Rahman 2025) dataset is shown below. The output from the **Query Entities Checker** is simplified for convenient visual inspection in the actual pipeline it returns results in a structured JSON format.

Q: How many characters have Corlys Velaryon as their father or are married to Daemon Targaryen?

Query Generator

```
MATCH (father:Character),
        (spouse:Character)
WHERE toLower(father.name) =
        'corlys velaryon'
AND toLower(spouse.name) =
        'daemon targaryen'
MATCH (c:Character)
WHERE (c)-[:hasFather]->(father)
        OR (c)-[:hasSpouse]->(spouse)
RETURN count(DISTINCT c)
```

Graph DB Executor

Memgraph returned the exception:

Not yet implemented: atom expression
(c)-[:hasFather]->(father).
Try to rewrite the query by using OPTIONAL
MATCH, WITH and WHERE clauses.

(Query execution attempts made: 1)

Query Evaluator

Evaluation: **Error or empty**

Feedback:

The query failed because it uses a pattern expression, such as (c)-[:hasFather]->(father), directly inside a **WHERE** clause. This syntax is not supported by Memgraph, as indicated by the error message. To check for the existence of a relationship pattern, it must be wrapped in the EXISTS() function. The query should be rewritten to use **OPTIONAL MATCH** or separate **MATCH** clauses with **WITH** to combine the results, which is a more robust pattern.

For example:

```
MATCH (child:Character)-[:hasFather]->(f:
        Character)
WHERE toLower(f.name) = 'corlys velaryon'
WITH COLLECT(child) AS children MATCH
        (spouse_char:Character)-[:hasSpouse]->(s:
        Character)
WHERE toLower(s.name) = 'daemon targaryen'
WITH children + COLLECT(spouse_char) as
        allPeople!
UNWIND allPeople as person
RETURN count(DISTINCT person).
```

Verification Module: Query Entities Extractor

Extracted entities:

```
{
  "node_labels": [
    "Character"
  ],
  "node_property_values": [[
    "Character",
    "name",
    "corlys velaryon"
  ], [
    "Character",
    "name",
    "daemon targaryen"
  ]
],
  "pairwise_relationships": [
    "(:Character)-[:hasFather]->(:
      Character)",
    "(:Character)-[:hasSpouse]->(:
      Character)"
  ]
}
```

Verification Module: Query Entities Checker

Node labels verification result:

null

Nodes properties verification results:

Node label: "Character"

Node property value used in the query:
"corlys velaryon"

Suggested node property corrections:

Property: "Corlys Velaryon"
Levenshtein similarity score: 86.66

Property: "Lucerys Velaryon"
Levenshtein similarity score: 77.41

Property: "Jacaerys Velaryon"
Levenshtein similarity score: 75.0

Other semantically relevant suggestions:

"Laenor Velaryon",
"Laena Velaryon",
"Jacaerys Velaryon"

Node label: "Character"

Node property value used in the query:
"daemon targaryen"

Suggested node property corrections

Property: "Daemon Targaryen"
Levenshtein similarity score: 87.5

Property: "Aemon Targaryen"
Levenshtein similarity score: 83.87

Property: "Aemond Targaryen"
Levenstein similarity score": 81.25

Other semantically relevant suggestions:
"Daemon Targaryen",
"Aemond Targaryen",
"Daeron Targaryen"

Relationships verification results:

[]

Verification Module: Instructions Generator

Based on the verification results, here are the required corrections for your Cypher query:

1. In the Character node, the property value 'corlys velaryon' is incorrect. Replace it with 'Corlys Velaryon'.
2. In the Character node, the property value daemon targaryen is incorrect. Replace it with Daemon Targaryen.

Feedback Aggregator

Your query incorrectly used an expression in the **WHERE** clause. Instead, use two separate **MATCH** clauses for each condition: one to find characters :hasFather Corlys Velaryon, and another to find characters :hasSpouse Daemon Targaryen. Combine the results from both matches, then unwind the list and return the count of distinct characters. Ensure you use the exact, case-sensitive names Corlys Velaryon and Daemon Targaryen for the 'name' property.

(Query generation attempt: 2)

Query Generator

```
OPTIONAL MATCH
(child:Character)-[:hasFather]->
(:Character{name: 'Corlys Velaryon'})
WITH collect(DISTINCT child) AS children
OPTIONAL MATCH
(spouse_char:Character)-
[:hasSpouse]-
(:Character{name: 'Daemon Targaryen'})
WITH children + collect(
DISTINCT spouse_char
)
AS all_chars UNWIND all_chars AS c
RETURN count(DISTINCT c)
```

Graph DB Executor

Query outcome:

```
[
  {
    "count(DISTINCT c)": 3
  }
]
```

Query Evaluator

Evaluation: Accept

Feedback:

The query correctly implements the required logic. It uses two separate **OPTIONAL MATCH** clauses to find characters satisfying each condition (child of Corlys, spouse of Daemon), combines the results into a single list, and then counts the distinct characters in that list. This is a valid, though somewhat complex, way to implement an **OR** condition and correctly answers the user's question.

Interpreter

Answer:

There are 3 characters who have *Corlys Velaryon* as their father or are married to *Daemon Targaryen*.

Appendix C

Below are the predicted answers and generated corresponding Cypher queries on the Sample House IFC data using our Multi-Agent GraphRAG pipeline. The user questions and corresponding expected answers are sourced from (Iranmanesh, Saadany, and Vakaj 2025).

User question	Expected response	Generated Cypher query	Given response
1. How many doors exist in the building?	There are 3 doors.	<pre>MATCH (d:IfcDoor) RETURN count(d)</pre>	The building contains 3 doors.
2. What is the volume of the roof space?	The gross volume of the roof space is 76.47 cubic meters.	<pre>MATCH (s:IfcSpace) WHERE s.Constraints.Level = 'Roof' RETURN s.BaseQuantities.GrossVolume</pre>	The gross volume of the roof space is 76465.52 m³. This value appears unusually large.
3. How many storey exist in the building?	There are 2 storey in the building	<pre>MATCH (n:IfcBuildingStorey) RETURN count(n)</pre>	The building contains 2 stories.
4. What is the gross floor area of the entrance hall?	The gross floor area of the entrance hall is 8.69 square meters.	<pre>MATCH (space:IfcSpace) WHERE toLower(space.Name) CONTAINS 'entrance hall' RETURN space.Name, space.BaseQuantities.GrossFloorArea</pre>	The gross floor area of the space '3 - Entrance hall' is 8.69 m ² .
5. What is the illuminance unit defined in the file?	The illuminance unit defined in the file is Lux.	<pre>MATCH (unit:IfcSIUnit) WHERE unit.UnitType = 'ILLUMINANCEUNIT' RETURN unit.Name</pre>	The defined unit for illuminance is LUX.
6. Is there a Laundry in the building?	No there isn't any Laundry space in the building.	<pre>MATCH (s:IfcSpace) WHERE toLower(s.Name) CONTAINS 'laundry' OR toLower(s.LongName) CONTAINS 'laundry' OR toLower(s.IdentityData.Name) CONTAINS 'laundry' RETURN count(s) > 0</pre>	No, the database contains no space identified as a Laundry
7. What is the perimeter of the entrance hall?	The entrance hall has a perimeter of 12,810.	<pre>MATCH (s:IfcSpace) WHERE toLower(s.Name) CONTAINS 'entrance hall' RETURN s.Name, s.BaseQuantities.GrossPerimeter</pre>	The gross perimeter of the space '3 - Entrance hall' is 12.81 m.
8. What is the building address?	The address is 'Westminster, London, UK'	<pre>MATCH (b:IfcBuilding) -[:BUILDINGADDRESS]-> (pa:IfcPostalAddress) RETURN pa.AddressLines, pa.Town, pa.Region, pa.Country</pre>	The building address is Westminster, London, UK. Some fields are missing.
9. What is the unconnected height of the roof space?	The roof has an unconnected height of 1000 mm (1 meter).	<pre>MATCH (s:IfcSpace) WHERE s.Constraints.Level = 'Roof' RETURN s.Dimensions.UnboundedHeight</pre>	The unconnected height of the roof space is 1.00 m.
10. What is the name of the project?	The project name is 'ProjectName'.	<pre>MATCH (p:IfcProject) RETURN p.LongName</pre>	The project is named Project Name.