CS-E4780 Scalable Systems and Data Management Course Project Enhancing LLM Inference with GraphRAG

Linh Van Nguyen
Aalto University
Espoo, Finland
linh.10.nguyen@aalto.fi

Prof. Bo Zhao*
Aalto University
Espoo, Finland
bo.zhao@aalto.fi

Abstract

Graph Retrieval-Augmented Generation (GraphRAG) promises faithful question answering over richly connected data, yet practical deployments demand careful control of query synthesis, validation, and evaluation latency. This report presents an enhanced GraphRAG pipeline for the Nobel Laureates benchmark that advances the course baseline along all required dimensions. We design a schema-driven Text2Cypher module with iterative repair, implement Pydantic-based type validation for agent interactions, and introduce comprehensive runtime instrumentation. Evaluations across five state-of-the-art language models on a manual building benchmark. A two-phase latency analysis shows that query synthesis dominates runtime but remains tractable for interactive use. The resulting system delivers precise, reproducible answers and provides a template for future GraphRAG deployments in academic settings. Github: github.com/linhnguyen1012/GraphRAG_Nobel_Prize.git

Keywords

GraphRAG, Text2Cypher, retrieval-augmented generation, Pydantic, multi-process evaluation

1 Introduction

Large Language Models (LLMs) struggle to reason over richly connected data when only textual memories are available. Graph Retrieval-Augmented Generation (GraphRAG) systems [1] mitigate this limitation by coupling LLMs with graph databases that encode entities and relations explicitly. For the CS-E4780 project we focus on the Nobel Laureates dataset provided with the Kùzu graph database [2], aiming to produce precise, explainable answers to multihop questions.

Our contribution include an end-to-end GraphRAG workflow that advances the course baseline through major steps:

(1) **Schema-driven prompt engineering.** We employ an LLM agent to prune the full graph schema down to only the nodes, edges, and properties relevant to each question, ensuring the Text2Cypher model receives focused context without unnecessary noise.
(2) **Self-refining Cypher generation with validation feedback.** A repair loop validates candidate queries via `EXPLAIN` dry-runs and injects targeted feedback when syntax or semantics fail, allowing up to three repair attempts per query.
(3) **Pydantic-based type validation.** We enforce structural contracts on all agent inputs and outputs using Pydantic schemas, preventing malformed data from propagating through the pipeline and enabling safe multi-process evaluation [3].
(4) **Comprehensive two-level evaluation.** We implement a dual assessment strategy that separately measures query grounding (whether Text2Cypher retrieves the correct context) and answer correctness, using an LLM judge to handle flexible natural-language responses.

Our design choices are informed by recent progress in retrieval-augmented generation and graph-based reasoning [1]. We explicitly target the tasks mandated by the CS-E4780 project brief: extending Text2Cypher, introducing caching, and reporting fine-grained latency. Beyond strict requirements, we emphasize reproducibility and maintainability, as experience shows that LLM-enabled systems degrade quickly without strong interfaces and evaluation harnesses.

2 Background

Graph Retrieval-Augmented Generation (GraphRAG) addresses the limitations of traditional RAG systems by leveraging structured graph databases instead of unstructured text corpora. While conventional RAG retrieves relevant passages through vector similarity, GraphRAG exploits explicit entity and relationship representations encoded in property graphs, enabling precise multi-hop reasoning over connected data. This approach proves particularly effective for domains with complex relational structures, such as knowledge bases, citation networks, and biographical databases.

The Nobel Laureates dataset exemplifies such a domain: scholars are connected to prizes, institutions, geographic entities, and each other through richly typed relationships (`WON`, `AFFILIATED_WITH`, `BORN_IN`, `MENTORED`). Answering questions like "Which laureates from Finland?" requires traversing multiple relationship types—a task poorly suited to keyword-based text retrieval but natural for graph query languages like Cypher.

The central challenge in GraphRAG systems lies in translating natural language questions into executable graph queries. Early approaches relied on rule-based parsers or trained sequence-to-sequence models, both requiring extensive domain-specific engineering or large labeled datasets. Recent work demonstrates that large language models can generate graph queries from natural language when provided with schema context and few-shot examples [1]. However, LLMs frequently produce syntactically invalid or semantically incorrect queries, motivating iterative refinement strategies that validate and repair candidates before execution.

3 System Architecture

Our pipeline comprises seven modules:

*Supervisor. Responsible Teacher

CS-E4780, Espoo, Finland
2025.

Table 1: Mapping between course requirements and implemented features.

Requirement	Implementation highlight
Schema-driven prompting	LLM agent prunes full schema to relevant nodes/edges/properties per question
Self-refinement loop	Iterative `EXPLAIN` validation with error-conditioned prompts
Rule-based conventions	Syntax rules embedded in system prompts (lowercase, CONTAINS, property naming)
LRU caching	Two-tier cache for questions and query plans (results reported qualitatively)
Stage-wise latency	Instrumented timers aggregated in Table 5
Performance visualization	Lightweight notebooks consume structured run logs to render flame graphs
Evaluation pipeline	Pydantic-typed judge harness with multi-process execution

- **Schema projection.** LLM agents summarizes Kùzu's Nobel schema and filters nodes/edges relevant to each question. The agent enforces property usage guidelines, e.g., matching laureate names on `knownName`.
- **Prompt engineering.** System prompts encode domain-specific rules and syntax constraints (e.g., property matching conventions, lowercase comparisons) to guide the Text2Cypher model without additional training.
- **Text2Cypher generation.** Prompt templates incorporate the pruned schema and syntax rules. The generator produces a Cypher candidate, which immediately runs through syntactic validation via `EXPLAIN`. Failures trigger repair instructions describing the error message and previously attempted queries.
- **Execution and caching.** Validated queries execute against Kùzu. We maintain an LRU cache keyed by {question hash, schema hash}, but omit cache hits from reported metrics so that model comparisons remain fair.
- **Answer synthesis.** The retrieved records feed an answer LLM that formats natural language responses. When context is empty the model states the insufficiency explicitly, avoiding hallucinations.
- **LLM judge.** A separate evaluator model grades both the retrieved context (query grounding) and the final response. Evaluation relies on structured rubrics that return `true`, `false`, or `no_attempt`.
- **Benchmark harness.** The workflow records intermediate artefacts in structured run archives so that experiments can be re-run and audited.

3.1 Schema Understanding and Prompting

We treat schema understanding as an independent agent because multihop queries frequently rely on multiple relationship types. The agent consumes a serialized schema graph and the natural-language question, then emits a pruned schema graph containing only relevant node and edge labels. This output is validated against a Pydantic model to ensure, for instance, that all referenced properties exist and that relationship directions are preserved. By constraining the prompt to these schema fragments we reduce token usage and steer the Text2Cypher model toward semantically coherent patterns.

3.2 Iterative Text2Cypher with Repair Feedback

The Text2Cypher generator operates in an iterative refinement loop. After each generation attempt, we validate the candidate query by running `EXPLAIN` against the Kùzu database. Syntax or planning errors trigger a new attempt that includes the previous query, the error message, and explicit repair guidance enumerating what went wrong (e.g., "property `knownName` must be lower-cased before comparison," "missing WHERE clause for string matching"). This feedback mechanism allows the model to learn from its own mistakes within the conversation context. In practice, most queries converge within three attempts, and we cap the loop to avoid runaway costs. When all retries are exhausted without producing a valid query or when valid queries return empty results, the system returns an explicit "insufficient information" response rather than hallucinating.

3.3 Execution, Caching, and Answering

Valid queries flow through the execution layer, which pushes parameterized Cypher strings to the Kùzu engine. Query results are normalized into dictionaries keyed by entity labels and property names, enabling consistent downstream consumption. We deploy a two-level LRU cache: an upstream cache maps question hashes to schema summaries, and a downstream cache stores validated query plans. While we do not report cache hit ratios in the main results (per instructor guidance), qualitative use indicates that paraphrased questions benefit immediately from prior runs.

The answer synthesis module consumes the question, retrieved context, and executed results to draft a natural-language response. We purposely keep the format free-form to allow the model to justify reasoning steps when appropriate. The module also exposes structured metadata (e.g., which triples supported the answer) that the judge can inspect.

3.4 Evaluation Harness

Finally, the evaluation harness compares system outputs against gold answers. It aggregates metrics in two buckets: **query grounding fidelity** (does the Text2Cypher pipeline retrieve the requisite supporting facts?) and **answer correctness** (does the final answer match the gold answer?). Judge outputs are stored alongside raw responses, enabling manual auditing. All harness inputs and outputs satisfy Pydantic schemas, which prevents malformed evaluations and makes it straightforward to swap in alternative judges.

4 Implementation

The system is implemented using the official Kùzu Python bindings for graph access. Prompt management and agent wiring leverage minimal custom code instead of heavyweight frameworks, while Pydantic models ensure every inter-agent message satisfies structural constraints [3]. This design guards against schema drift as new features are added.

To parallelize evaluation we employ `multiprocessing` with spawn semantics, allowing independent LLM clients to execute without sharing sockets. A task scheduler batches queries per model configuration and records wall-clock timings for each pipeline stage. The experiments ran on a dual-socket Intel Xeon E5-2670 server (2 sockets × 8 cores × 2 threads = 32 logical CPUs @ 2.60 GHz base, 3.30 GHz turbo; 40 MB L3 cache) running Linux. Cloud-hosted LLM endpoints include OpenAI `gpt-5`, `gpt-5-mini`, `gpt-4.1`, `gpt-4.1-mini`, and Google `gemini-2.5-flash`. We cap retry attempts at either one or three depending on the ablation.

Benchmark data consist of nine manually curated question–answer pairs sourced from Nobel Prize archives and Wikipedia. Each base question is paraphrased to create additional surface forms, stressing the breadth of schema interactions and the cache's ability to disambiguate near-duplicates. Gold answers are reviewed for correctness and stored alongside the expected context triples.

4.1 Software Stack and Tooling

We intentionally keep the infrastructure lightweight. Dependency management relies on standard Python packaging, while configuration is controlled by typed data models implemented with Pydantic. Each agent (schema predictor, Text2Cypher, answerer, judge) exposes a common interface so the same code serves interactive experiments and offline evaluation.

All experiments run through a single entry point that consumes structured configuration files describing the model lineup, retry budget, and cache settings. The configuration is validated before execution, eliminating ambiguous command-line flags. Completed runs archive their configurations, logs, and metrics under timestamped directories to ensure reproducibility.

4.2 Prompt Engineering Toolkit

Prompt templates live alongside their schema definitions. A lightweight templating layer inserts dynamic content such as schema fragments and prior failed attempts with error messages. Each template encodes syntax rules (e.g., lowercase string matching, `knownName` property usage) that act as rule-based post-processors embedded in the generation prompt rather than as separate code modules. We also implement a prompt checksum mechanism: before sending a prompt to an LLM, we hash its contents and record the digest alongside the response. This makes it easy to detect when seemingly identical runs diverge because of subtle prompt drift.

4.3 Logging, Telemetry, and Reproducibility

Runtime telemetry is captured as structured event logs containing timestamps, stage identifiers, latency measurements, and outcome codes. A thin wrapper adds correlation identifiers so we can stitch together events across processes. These traces drive both the aggregated tables reported in Section 5 and qualitative analysis notebooks. For performance visualization we export the traces to standard timeline viewers, enabling detailed flame graphs without additional instrumentation.

To support collaborative debugging, we provide a replay command that regenerates a logged run by rehydrating prompts, schema snapshots, and cached responses. This capability proved invaluable when verifying that the repair loop behaved consistently across hardware environments.

Table 2: Overall accuracy and precision across retry budgets. Bold values denote the best score per column.

Model	Acc@1	Prec@1	Acc@3	Prec@3
gemini-2.5-flash	0.68	0.91	0.79	0.92
gpt-5	0.69	0.89	0.83	0.88
gpt-5-mini	**0.83**	**0.97**	0.89	0.91
gpt-4.1	0.76	0.93	0.89	**0.94**
gpt-4.1-mini	0.64	0.88	**0.91**	**0.94**

Table 3: Query grounding accuracy across retry budgets. Values are rounded to two decimal places.

Model	Grounding@1	Grounding@3
gemini-2.5-flash	0.65	0.64
gpt-5	0.56	0.58
gpt-5-mini	0.72	0.81
gpt-4.1	0.58	0.71
gpt-4.1-mini	0.50	0.82

5 Evaluation

5.1 Experimental Setup

Each model-answer configuration is evaluated on the 27-query benchmark under two retry budgets. Metrics are computed from LLM judge outputs: **accuracy** counts correct answers among all attempts, while **precision** measures correctness conditional on producing an answer. For fairness, cache hits are disabled during evaluation, but the cache remains active in production deployments.

5.2 Accuracy and Precision Results

Table 2 summarizes overall accuracy and precision for both retry settings. Increasing the retries from one to three improves accuracy for every model, and the gap is largest for `gpt-4.1-mini`, which benefits from additional repair opportunities. Precision remains high across the board, indicating that aggressive retries do not introduce spurious answers.

5.3 End-to-end Runtime Across Models

Timers embedded in the evaluation harness also capture the wall-clock time per query for each model configuration. Table 4 summarizes the mean latencies under the two retry budgets. Moving from one to three retries increases runtime modestly for the smaller models, adding about one second for `gpt-4.1-mini`, while the larger

Table 4: Average per-query runtime (seconds) across models and retry budgets.

Model	Time@1 (s)	Time@3 (s)
gemini-2.5-flash	17.3	20.2
gpt-5	53.9	75.4
gpt-5-mini	32.4	37.3
gpt-4.1	8.0	9.5
gpt-4.1-mini	7.0	8.0

Table 5: Stage-level latency breakdown for `gpt-4.1-mini` at three retries. Measurements averaged over the 27-query benchmark.

Pipeline Phase	Avg Time (s)
Query synthesis (Text2Cypher + validation)	4.8
Answer resolution (execution + response)	3.2
Total	**8.0**

`gpt-5` model incurs a steeper penalty. These measurements inform the recommendations in Section 6.

5.4 Latency Breakdown

To highlight where optimizations matter, we instrumented the best-performing configuration (`gpt-4.1-mini` with three retries) and averaged stage-level latencies over the 27-query benchmark. Timers embedded in the evaluation harness aggregate two macro phases: (i) **query synthesis**, covering schema projection, Text2Cypher drafting, and dry-run validation with iterative repair; and (ii) **answer resolution**, covering graph execution, response drafting, and judging. Table 5 reports the resulting averages. The synthesis phase dominates runtime, suggesting that further engineering should target prompt efficiency or partial reuse of validated subgraphs.

5.5 Ablation: Retry Budget Versus Latency

We observe varying latency and accuracy trade-offs across the five models when raising the retry cap from one to three. For `gpt-4.1-mini`, the mean per-query runtime grows from 7.0 s to 8.0 s while accuracy increases from 0.64 to 0.91 (a 27% absolute gain). Larger models such as `gpt-5` incur a steeper latency penalty (54 s to 75 s) with more modest accuracy improvement (0.69 to 0.83), suggesting that model-specific retry policies or adaptive early stopping based on validation confidence could optimize the accuracy-latency trade-off. Future work could explore speculative execution: generating multiple query candidates in parallel and validating them asynchronously to reduce wall-clock time.

6 Discussion

Three insights emerge from the evaluation. First, even lightweight frontier models such as `gpt-4.1-mini` can match the accuracy of larger models when afforded multiple retries, underscoring the value of self-refinement. Second, the latency breakdown reveals that database execution is not the bottleneck; instead, LLM interactions dominate, motivating techniques like prompt distillation or speculative decoding. Third, while caching is not included in the reported metrics, qualitative inspection confirms that the LRU caches eliminate redundant schema queries and stable paraphrases in interactive settings.

Building on these findings, we recommend three operating modes. For accuracy-oriented deployments, pair `gpt-4.1-mini` as the Text2Cypher specialist with `gpt-4.1` for answer generation, enabling strong grounding (0.82) and high precision (0.94) under the three-retry budget while keeping latency near 8–9 seconds per query. For latency-sensitive scenarios, a single retry with `gpt-4.1-mini` delivers responses in roughly 7 seconds at the cost of lower grounding. Finally, when budget permits heavier models, `gpt-5` benefits most from extra retries, but its 75-second runtime makes it best suited for offline batch processing rather than interactive use.

Our observations align with broader trends in retrieval-augmented generation: the retrieval stack (schema selection, query formulation) contributes more to overall quality than marginal improvements in the answer-generation model once that model crosses a competency threshold. Consequently, future work should prioritize systematic retrieval diagnostics, a broader benchmark design, and automated repair strategies. The structured logging infrastructure we built lays the groundwork for such efforts by capturing every intermediate artefact.

Limitations include the small benchmark size and reliance on LLM judges, which may exhibit grading variance. Expanding the benchmark with community contributions and incorporating human validation would strengthen future analyses. We also note that the LLM APIs used in this study evolve rapidly; re-running the evaluation months later may yield different results due to backend model updates. Freezing prompts, random seeds, and manifests mitigates the issue but does not eliminate it. Finally, integrating access control and auditing remains future work before deploying the system in broader academic settings, where provenance tracking and privacy assurances are critical.

7 Conclusion

We delivered a robust GraphRAG pipeline that improves answer quality on Nobel laureate queries through schema-driven prompting, iterative self-correction, and rigorous LLM-based evaluation. The combination of Pydantic type validation and multi-process execution keeps the system maintainable while enabling rapid experimentation. Looking forward, we plan to test structured decoding methods for Cypher generation, integrate cost-aware model routing, and investigate more expressive caching strategies that operate at the subgraph level.

Detailed prompt templates, benchmark definitions, and extended quantitative results are provided in the appendices to support reproducibility and reuse.

Acknowledgments

Thanks to **Professor Bo Zhao** for the insightful course assignment and the valuable exercise sessions, which provided significant guidance and motivation for this project. I also want to thank to the

Teaching Assistants for establishing the foundational framework and resources that served as the starting point for this work.

In this course, I completed a solo project, covering all stages from literature survey and implementation to the final report.

This study has two primary limitations. First, the retrieval mechanism relies exclusively on graph-based Text2Cypher generation. We have not yet implemented a **hybrid retrieval strategy** that combines this structured approach with semantic (e.g., vector-based) retrieval. Such a hybrid system could improve robustness, especially for ambiguous queries that do not map cleanly to the graph schema.

Second, the evaluation is constrained by the **scale of our benchmark**. The current dataset, which consists of **27 manually constructed question-answer pairs**, is sufficient for initial validation but may not represent the full spectrum of potential query complexity. Future work should involve validation against a larger and more diverse standardized benchmark.

Several practical challenges were encountered during this project. The first significant hurdle was the **lack of an existing, realistic benchmark** for this specific domain, which required the substantial effort of manually constructing and validating the 27-query benchmark.

The second major challenge was **identifying models capable of high performance** on this task. We found that generating syntactically correct and semantically accurate Cypher is non-trivial. Initial experiments with **self-hosted open-source models, including variants with up to 20 billion parameters**, yielded unsatisfactory results, struggling with schema adherence and complex logic. This bottleneck ultimately guided our selection toward the more advanced proprietary models evaluated in this paper.

A Prompt Templates

Listing 1: Schema projection system prompt.

```
schema_agent_prompt = """
Understand the given labelled property graph schema and the given user question.
Your task is to return ONLY the subset of the schema (node labels, edge labels and
properties) that is relevant to the question.

- The schema is a list of nodes and edges in a property graph.
- The nodes are the entities in the graph.
- The edges are the relationships between the nodes.
- Properties of nodes and edges are their attributes, which helps answer the question.

Return a JSON object with 'nodes' and 'edges' arrays matching the GraphSchema structure.
"""
```

Listing 2: Text2Cypher system prompt with repair scaffolding.

```
text2cypher_prompt = """
Translate the question into a valid Cypher query that respects the graph schema.

<SYNTAX>
- When matching on Scholar names, ALWAYS match on the `knownName` property.
- For countries, cities, continents and institutions, match on the `name` property.
- Use short, concise alphanumeric strings as variable bindings (e.g., a1, r1).
- Respect relationship directions using the schema information.
- When comparing string properties:
    * Lowercase both sides before comparison.
    * Use the WHERE clause with CONTAINS for substring checks.
- DO NOT use APOC extensions.
</SYNTAX>

<RETURN_RESULTS>
- Return integers as integers (not strings).
- Return property values instead of entire nodes.
- Do not coerce data types.
- The output MUST be a JSON object with a single-line 'query' field.
</RETURN_RESULTS>
"""
```

Table 6: Key configuration parameters for evaluation runs.

Parameter	Value
Hardware	Dual Intel Xeon E5-2670 (32 logical CPUs @ 2.60–3.30 GHz, 40 MB L3)
Database	Kùzu 0.8.1, Nobel Laureates snapshot (Oct 2025)
Prompt engineering	Rule-based syntax constraints embedded in system prompts
LLM endpoints	OpenAI `gpt-5`, `gpt-5-mini`, `gpt-4.1`, `gpt-4.1-mini`; Google `gemini-2.5-flash`
Retry budgets	{1, 3}, with early exit on first successful validation
Cache policy	Two-level LRU (question hash, schema hash); disabled for evaluation metrics
Judge rubric	Binary rubric for query grounding and final answer correctness
Logging format	Structured event logs with monotonic timestamps and correlation IDs

Listing 3: Fallback repair prologue injected after failed attempts.

```
def make_repair_section(previous_attempts: list[Attempt]) -> str:
    if not previous_attempts:
        return ""
    prompt = ["### Previous Failed Attempts ###"]
    for i, attempt in enumerate(previous_attempts):
        prompt.append(f"\nAttempt {i}:")
        prompt.append(f"\nGenerated Query: {attempt.query}")
        prompt.append(f"\nError/Issue: {attempt.error}")
    prompt.append("\nPlease learn from these failures and generate a corrected query that addresses all issues above.")
    return "\n".join(prompt)
```

References

[1] Darren Edge, Jonathan Wang, Augustin Boursier, Haitao Zheng, Aaron Baugh, and Michael M. Mathews. 2023. GraphRAG: Unlocking Retrieval-Augmented Generation for Knowledge Graphs. Microsoft Research Blog. Retrieved November 17, 2025 from https://arxiv.org/html/2501.00309v2.

[2] CS-E4780-project2. Githubhttps://github.com/benyucong/CS-E4780-project2.

[3] Samuel Colvin, Morgan Flores, and the Pydantic Contributors. 2024. Pydantic: Data Validation and Settings Management Using Python Type Annotations. Version 2.7.3. https://docs.pydantic.dev/.