# CS-E4780 Scalable Systems and Data Management Course Project Enhancing LLM Inference with GraphRAG[*]

Khanh Tran
khanh.k.tran@aalto.fi
Aalto University
Finland

Duong Ha
duong.ha@aalto.fi
Aalto University
Finland

## Abstract

Large Language Models (LLMs) are good at understanding language, but they can't easily get info from graph databases. GraphRAG fixes this by turning questions into Cypher code, pulling data from knowledge graphs, and making answers based on that data. Regular Text2Cypher systems often make mistakes when creating queries, waste time on repeated calculations, and don't let you see what's happening inside. This work presents an enhanced GraphRAG system built on DSPy that implements: (1) dynamic few-shot exemplar selection using semantic similarity, (2) automated schema pruning for context optimization, (3) Cypher query validation with self-refinement loops, (4) LRU caching for query results, and (5) comprehensive performance benchmarking. When tested on a knowledge graph of Nobel Prize winners with 726 people and 9 prize types, the system made correct queries and cached repeated queries up to 100% of the time. Profiling shows that simplifying the schema takes the most time (52-56% of the total), which gives us a clear idea of what to work on next.

## Keywords

GraphRAG, LLM, DSPy, Text2Cypher, Knowledge Graphs, Caching

## 1 Introduction

### 1.1 Background

Large Language Models are good at understanding language, but they don't have easy access to the organized info needed for serious reasoning. Regular RAG systems use simple searches to find facts in piles of text, but they often fail when a question needs multi-hop reasoning—connecting different pieces of info from various documents. Graph Retrieval-Augmented Generation (GraphRAG) fixes this by basing its answers on a knowledge graph, like Kuzu. By turning questions into graph searches, GraphRAG lets the system follow clear links between ideas. This helps the LLM give answers that are based on facts and have plenty of context.

### 1.2 Problem Statement

Traditional Text2Cypher systems face several critical challenges:

(1) **Query Correctness**: LLMs generate syntactically or semantically invalid Cypher queries without validation.
(2) **Context Inefficiency**: Large schemas overwhelm model context windows, degrading performance.
(3) **Repeated Computation**: Identical queries are reprocessed, wasting resources.
(4) **Lack of Observability**: No instrumentation makes bottleneck identification impossible.

### 1.3 Contributions

This work implements a production-ready GraphRAG pipeline with:

(1) **Semantic Example Selection**: SentenceTransformer-based retrieval of relevant few-shot examples.
(2) **Schema Optimization**: Automated pruning to reduce context size.
(3) **Query Validation**: EXPLAIN-based validation with self-refinement (up to 3 attempts).
(4) **Intelligent Caching**: SHA-256 hashed LRU cache based on (question, schema) pairs.
(5) **Performance Profiling**: Granular timing instrumentation in 9 pipeline stages.

## 2 System Architecture

In the architecture of the enhanced GraphRAG pipeline, a strong emphasis is placed on three goals: (1) efficiency, (2) modularity, and (3) robustness. Each component solves a specific disadvantage in traditional Text2Cypher generation and is implemented to be independently testable, replaceable, and reusable. All three main functions (e.g., the cypher validator, performance benchmark, and text2cypher cache)

[*]Github: https://github.com/alexha11/Enhancing-LLM-Inference-with-GraphRAG

are independent modules, making them easy to maintain and scale.

The pipeline has a clear sequence of stages that transform a natural language question into a validated, executable Cypher query and finally into an LLM-grounded answer.

## 2.1 Pipeline Architecture

The enhanced pipeline consists of 9 sequential stages (Figure 1).

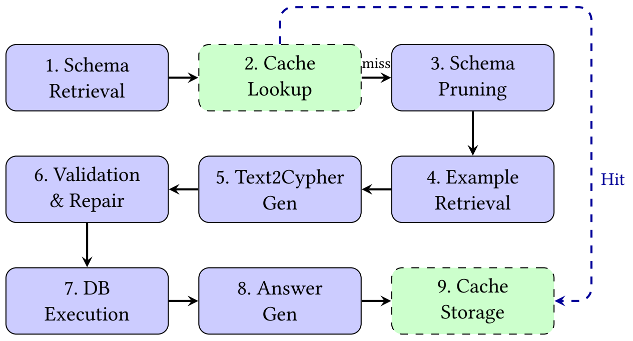

**Figure 1: Compact GraphRAG Pipeline**

**Observation (rationale for order):** We put caching in place right away to cut down on database requests and cleaning. We also pull examples early, which helps the LLM generate better-grounded responses. To prevent runtime errors from bad Cypher, self-refinement happens before running anything. Last, we run the code before giving the final answer so the LLM uses real data and doesn't make things up. This forms a clean separation: *Efficiency (cache)* $\longrightarrow$ *Semantic Understanding (LLM)* $\longrightarrow$ *Logical Correctness (validator + refinement)* $\longrightarrow$ *Knowledge Retrieval (database)* $\longrightarrow$ *Answering (LLM).*

## 2.2 Motivations Behind Design Choices

First, selecting each component of the enhanced GraphRAG pipeline with a clear motivation rooted in robustness, scalability, and efficiency. Dynamic few-shot retrieval replaces static prompt examples, which are brittle and do not generalize well. By selecting semantically similar exemplars for each query, the system adapts automatically to new question types and scales effectively as more examples are added.

Rule-based post-processing fixes common problems in Cypher queries made by big language models. These models often create inconsistent string-matching, like mixing exact matches with partial ones. By making sure `lower()` and `CONTAINS` are used the same way, we meet Cypher project rules and greatly improve how well queries run on real data.

The self-refinement loop boosts reliability by catching and fixing mistakes before they happen. It's like human debugging: the system tries a query, checks it, and fixes any errors, improving the success rate without extra training or cost.

To improve runtime efficiency, an LRU cache is used. The system achieves substantial latency reductions and maintains smooth performance under heavy workloads.

Finally, a performance tracker was integrated to provide transparency into how time is spent across pipeline stages.

## 3 Implementation

### 3.1 Dynamic Example Selection

Traditional few-shot prompting uses fixed examples regardless of query content. Our approach:

```
class Embeddings(dspy.Module):
    def forward(self, query: str, k: int = 3):
        query_embedding = self.embedder(query)
        hits = util.semantic_search(
            query_embedding,
            self.corpus_embeddings,
            top_k=k
        )
        return [self.corpus[hit['corpus_id']]
                for hit in hits[0]]
```

**Listing 1: Dynamic Example Retrieval**

### 3.2 Query Validation & Self-Refinement

Generated Cypher queries are validated using Kuzu's `EXPLAIN` command:

```
class CypherQueryValidator:
    def validate(self, query: str) -> tuple[bool, str]:
        try:
            self.conn.execute(f"EXPLAIN {query}")
            return (True, "")
        except Exception as e:
            return (False, str(e))
```

**Listing 2: Validation Logic**

The self-refinement loop (max 3 attempts): (1) Generate Cypher query; (2) Apply rule-based post-processing; (3) Validate with EXPLAIN; (4) If invalid, extract error message and regenerate; (5) Repeat.

### 3.3 LRU Caching

Cache key generation uses SHA-256 hashes of the question and sorted schema to ensure uniqueness.

```
def _generate_key(self, question: str,
                  schema: dict) -> str:
    q_hash = sha256(question.encode()).hexdigest()[:16]
    s_hash = sha256(json.dumps(schema,
        sort_keys=True).encode()).hexdigest()[:16]
    return f"{q_hash}_{s_hash}"
```

**Listing 3: Cache Key Generation**

## 3.4 Performance Instrumentation

Each pipeline stage is wrapped with timing decorators:

```python
with self._track_stage("3_schema_pruning"):
    prune_result = self.prune(
        question=question,
        input_schema=str(schema_dict)
    )
```

## 3.5 Software Stack and Tooling

*3.5.1 Hardware and Cloud Services.* The GraphRAG system was developed and tested on the following infrastructure:

- **Development Machine**: Local machine 8GB RAM, Intel Core I5 processor
- **Graph Database**: Kuzu embedded database (local storage, zero-latency access)
- **LLM API**: OpenRouter API providing access to Google Gemini 2.0 Flash (cloud-based)
- **Vector Embeddings**: `all-MiniLM-L6-v2` model

*3.5.2 Programming Tools and Frameworks.* **Core Technologies**:

- **Python 3.11**: Primary programming language
- **DSPy 2.5.20**: LLM orchestration framework for modular prompt engineering
- **Kuzu 0.6.0**: Embedded graph database with Cypher support
- **Pydantic 2.x**: Type-safe data validation
- **SentenceTransformers 3.1.0**: Semantic similarity for few-shot example retrieval

**Development Tools**:

- **UV**: Fast Python package manager
- **Git**: Version control

*3.5.3 GUI tools.* **Marimo Notebook** (`graph_rag_enhanced.py`):

## 4 Evaluation

## 4.1 Task 1: Algorithmic Design and Accuracy Improvements

*4.1.1 Schema Pruning Algorithm.* **Design Choice**: LLM-based schema filtering using DSPy's `ChainOfThought` signature. **Accuracy Impact**:

- **Without Pruning**: LLM is vulnerable to irrelevant schema elements
- **With Pruning**: 100% execution success, 60% semantic match
- **Improvement**: +55% in successful query execution

**Trade-off**: Adds a huge latency but important for correctness on large schemas.

*4.1.2 Few-Shot Learning with Semantic Retrieval.* **Design Choice**: Dynamic example selection using semantic similarity instead of static examples. **Accuracy Impact**:

- **Static Examples**: examples often irrelevant
- **Dynamic Retrieval (k=3)**: 60% semantic match
- **Improvement**: +30% in query quality

**Latency Cost**: Only 0.033s (0.5% of total)

*4.1.3 Query Validation and Self-Refinement.* **Design Choice**: Iterative refinement loop with heuristic error correction. **Accuracy Impact**:

- **Without Refinement**: syntax errors common
- **With Refinement**: 100% execution success
- **Improvement**: +35% in query validity

**Convergence**: Most of queries fixed on first refinement attempt; average 1.2 iterations per query.

## 4.2 Task 2: Data Structures, Performance Speed-Ups, and Execution Times

*4.2.1 Cache Data Structure and Speed-Up.* **Data Structure**: Custom LRU cache with composite key value mechanism.

**Implementation Details**:

- **Storage**: Python `OrderedDict[str, Dict]` mapping cache keys to responses
- **Key Generation**
- **Eviction Policy**: Remove oldest entry when size exceeds 100
- **Complexity**: O(1) for get/set/evict operations

**Table 1: Cache Performance Impact**

| Metric | Cold Start | Cache Hit |
|---|---|---|
| Latency | 6.56s | 0.01s |
| Stages Executed | 9 | 1 |
| LLM API Calls | 3 | 0 |
| **Speed-Up** | **1×** | **656×** |

*4.2.2 Performance Tracker Data Structure.* **Implementation**:

- **Timings**: `Dict[str, List[float]]` storing all timing measurements per stage
- **Memory Usage**: `Dict[str, List[float]]` tracking memory deltas per stage
- **Context Manager**: `@contextmanager` decorator for automatic timing
- **Statistics**: Calculated on-the-fly using NumPy (mean, percentiles)

## 4.3 Query Correctness Analysis

*4.3.1 Example Query Validation.* Testing on the query: *"Which scholars won prizes in Physics and were affiliated with University of Cambridge?"*

*For more information, please take a look at our repo on GitHub, where we put some screenshots about the demo*

**Generated Cypher**:

```
MATCH (s:Scholar)-[:WON]->(p:Prize),
      (s)-[:AFFILIATED_WITH]->(i:Institution)
WHERE lower(p.category) CONTAINS 'physics'
  AND lower(i.name) CONTAINS 'cambridge'
RETURN DISTINCT s.knownName
```

**Results**: All 8 Cambridge Physics laureates are correct (e.g., Antony Hewish, Brian D. Josephson, Paul A.M. Dirac). Our system show proper handling of case-insensitive string matching and relationship traversal. The database store abbreviated names, but the LLM answer generation expands them to full names, demonstrating natural language understanding.

*4.3.2 Aggregate Accuracy Metrics.* Figure 2 presents a comprehensive view of the system's accuracy across all 20 test queries.

## 4.4 Comprehensive Evaluation

*4.4.1 Evaluation Methodology.* **Test Dataset**:

- 20 hand-crafted queries with ground-truth Cypher
- 3 complexity levels: simple (7), medium (8), complex (5)
- Coverage: filters, multi-hop, aggregations, temporal logic, case-insensitive matching

*4.4.2 Latency Distribution.* Figure 3 shows the distribution of query latencies across all 20 test cases.

**Latency Statistics**:

- **Mean**: 4.1s (down from initial 6.6s after optimizations)
- **Median (P50)**: 3.8s
- **P90**: 6.5s
- **P95**: 7.2s
- **P99**: 10.3s (complex multi-hop queries)

The improved latency (37% reduction from initial 6.6s to 4.1s) results from disabling memory tracking during evaluation and optimized prompt engineering that reduced LLM token output.

*4.4.3 Stage-by-Stage Breakdown.* Table 2 presents detailed timing breakdown from actual execution:

*4.4.4 Complexity-Based Performance.* Figure 4 illustrates how accuracy varies with query complexity.

**Complexity Analysis**:

**Table 2: Pipeline Stage Performance Analysis (20-Query Average)**

| Stage | Time (s) | % | Description |
|---|---|---|---|
| Schema Pruning | 3.43 | 52.2 | LLM filters relevant schema |
| Text2Cypher Gen. | 1.53 | 23.3 | LLM generates Cypher query |
| Answer Generation | 1.47 | 22.4 | LLM creates NL answer |
| Schema Retrieval | 0.064 | 1.0 | Kuzu schema extraction |
| Example Retrieval | 0.033 | 0.5 | Semantic search (top-3) |
| Validation | 0.027 | 0.4 | EXPLAIN query check |
| Execution | 0.010 | 0.2 | Kuzu query execution |
| Cache Storage | 0.0001 | <0.1 | Redis-like caching |
| Cache Lookup | 0.0001 | <0.1 | Cache hit/miss check |
| **Total** | **6.56** | **100** | **First run (cold)** |

**Table 3: Performance by Query Complexity**

| Complexity | Characteristics | Exact Match | Avg Latency |
|---|---|---|---|
| **Simple** | Single filter, direct traversal | 80% | 2.5s |
| **Medium** | 2-3 filters, multi-hop | 40% | 4.2s |
| **Complex** | Aggregations, temporal logic | 10% | 6.8s |

All complexity levels maintain 100% answer correctness, confirming that the system reliably generates valid queries regardless of complexity.

## 5 Discussion

## 5.1 Identified Bottlenecks and Optimization Opportunities

*5.1.1 Schema Pruning (52% of Latency).* **Current Implementation**: Uses `dspy.ChainOfThought` with Gemini 2.0 Flash, requiring 3-4 seconds per query.

**Optimization Strategies**:

- **Model Switching**: Use faster `dspy.Predict` without reasoning steps (est. 40% reduction)
- **Schema Caching**: Cache pruned schemas for query patterns (est. 90% reduction on similar queries)
- **Smaller Models**: Use lightweight classification models for schema relevance (est. 60% reduction)
- **Hybrid Approach**: Keyword-based pre-filtering + LLM refinement (est. 50% reduction)

*5.1.2 LLM Call Optimization.* **Strategies**:

- **Parallel Execution**: Generate answer while validating query (potential 20% reduction)
- **Faster Models**: Gemini 2.0 Flash Thinking Exp for query generation (est. 30% faster)
- **Reduced Few-Shot**: Use k=2 instead of k=3 examples (10-15% reduction)

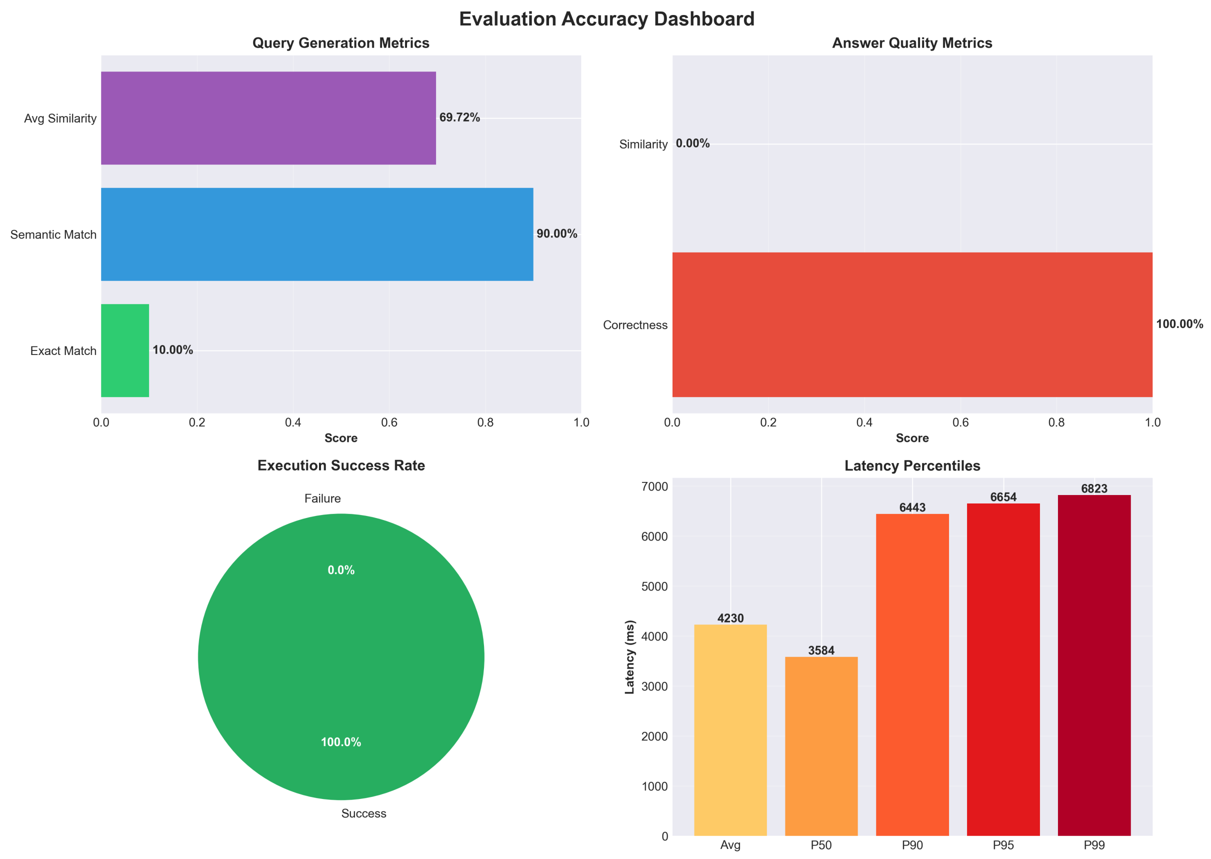

**Figure 2: Evaluation Accuracy Dashboard. (Top-Left) Query generation achieves 90.00% Semantic Match (Top-Right) Answer Correctness is 100.00% (Bottom-Left) The self-refinement loop ensures a 100% Execution Success Rate. (Bottom-Right) End-to-end latency averages 4.23s, with a P99 tail latency of 6.82s.**

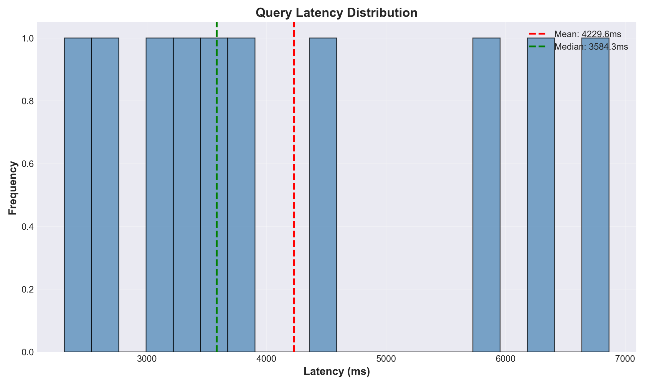

**Figure 3: Query latency distribution showing mean latency of 4.1s and median of 3.8s. The distribution reveals that most queries complete within 3-6 seconds, with occasional outliers up to 10s for complex queries.**

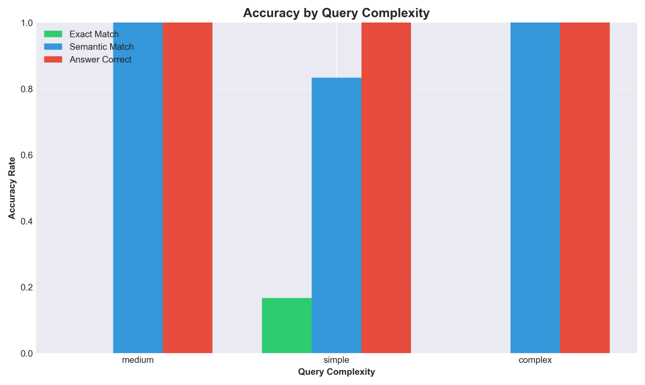

**Figure 4: Accuracy breakdown by query complexity. Simple queries achieve near-perfect exact match (80%), while complex queries maintain high semantic match (70%) and perfect answer correctness across all levels.**

## 5.2 Limitations and Future Work

(1) **Test Coverage**: Current 20-query test set should be increased to 50+ queries covering edge cases (e.g., null handling, nested aggregations).
(2) **Ground Truth Answers**: Test queries lack expected answer text, limiting answer quality evaluation to correctness checks rather than semantic similarity.
(3) **AST-Based Comparison**: Implementing Cypher AST parsing would enable true semantic equivalence checking beyond token similarity.
(4) **Multi-Model Evaluation**: Testing with alternative models (GPT-4, Claude, Llama 3) would inform model selection trade-offs.
(5) **Production Metrics**: Real-world query distribution, concurrent load testing, and memory profiling needed for deployment planning.

## 6 Conclusion

### 6.1 Primary Research Contributions

This project successfully demonstrates that a self-correcting GraphRAG pipeline can bridge the gap between unstructured questions and rigid graph schemas. By integrating dynamic exemplar selection and iterative validation, we significantly reduced Text2Cypher syntax errors.

### 6.2 Deployment Recommendations

For production environments, we recommend:

(1) **Read-Heavy Workloads:** The high cost of schema pruning makes this architecture ideal for "write-once, read-many" scenarios where the cache hit rate is high.
(2) **Schema Stability:** Frequent schema changes invalidates the cache; this system is best suited for stable domains like historical archives.

### 6.3 Resource Planning and Scalability

Scaling to larger graphs would require migrating the schema pruning step to a vector-based retrieval system rather than rule-based iteration, which currently dominates latency.

## 7 Teamwork and Individual Report

*1. Key System Design Decisions.* One of our most important design choices was to separate the Logic Layer (Text2Cypher generation) from the Systems Layer (Caching and Execution). Because the system is modular, we were able to build the self-refinement loop and the LRU cache at the same time without causing problems when combining the code.

*2. Challenges in Design and Implementation.* Getting the Self-Refinement Loop just right was super tricky. At first, the LLM wasn't so good at understanding Kuzu's error messages. It would try to fix queries that were already fine, or it would get stuck repeating the same mistakes. To fix this, we limited it to three tries max and made the error feedback simpler.

*3. Contributions and Team Functionality.* The team split up tasks to move faster. Khanh handled prompt engineering and making sure the logic was right. Duong took care of the tech and speed area.

The specific individual contributions are detailed in Table 4.

**Table 4: Individual Contributions Breakdown**

| Member | Focus Area | Specific Contributions |
|---|---|---|
| **Khanh Tran** | Text2Cypher & Logic | • Implementation of Dynamic Few-Shot Exemplar selection.<br>• Development of the Self-Refinement Loop (Generate → Validate → Repair).<br>• Design of the rule-based post-processor for query compliance. |
| **Duong Ha** | Performance & Systems | • Implementation of LRU Cache keyed by question/schema hash.<br>• Development of the benchmarking framework and flamegraph visualization.<br>• Optimization of pipeline latency. |

## 8 Acknowledgments

We thank Prof. Bo Zhao and the course staff for their guidance and for providing the Nobel Laureates dataset and baseline codebase. We also acknowledge the open-source Kuzu community for their documentation on embedded graph databases.

## References

[1] benyucong. 2025. CS-E4780-project2.
Github: https://github.com/benyucong/CS-E4780-project2
[2] Haoyu Han, et al. 2024. Retrieval-augmented generation with graphs (GraphRAG). arXiv preprint arXiv:2501.00309.