L'Oreal Capstone Knowledge Graph Construction Final Report

Brandon Keung (bk2951)[1], Jiaheng Dai (jd4136)[1], Jiayi Dong (jd4123)[1], and Pei Tian (pt2632)[1]

[1] *Columbia University, Fu Foundation School of Engineering and Applied Science*

Course:
ENGI E4800 Data Science Capstone & Ethics

Date:
May 19, 2025

Contents

Abstract

Large Language Models (LLMs) have transformed the way that people go about their daily lives. However, LLMs continue to face challenges such as hallucinations, factual accuracy, and domain-specific query resolution. Retrieval Augmented Generation (RAG) systems mitigate some of these issues by incorporating external domain knowledge. Classical RAG applications utilize vector based information to provide LLMs with meaningful information. However, graph-based RAG (GraphRAG) emerged as a promising alternative to the traditional vector based retreival methods. Research into GraphRAG shows promising results but challenges lie in constructing knowledge graphs (KG) and querying from them.

Our project investigates the impact of various KG construction techniques on GraphRAG performance, focusing on retrieval accuracy, response relevance, and inference speed. In this report we explore different KG construction ideas such as entity relationships and hierarchical structures to optimize graph formation. By evaluating KG methodologies, this research aim to advance GraphRAG's application in the e-commerce industry.

1 Introduction

1.1 Previous Work & Motivation

In recent years, Large Language Models (LLMs) have revolutionized natural language processing with their ability to generate coherent, contextually relevant responses. However, these models still face significant challenges related to hallucinations, factual accuracy, and contextual relevance when dealing with domain-specific queries. To address these limitations, Retrieval Augmented Generation (RAG) has emerged as a promising solution by augmenting LLMs with external knowledge sources. First introduced in the paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks" (Lewis et al., 2020), RAG systems enhance model outputs by retrieving and incorporating relevant information from external databases during the generation process.

Last semester's capstone project conducted a comprehensive comparison between traditional Vector-based RAG systems and Graph-based RAG (GraphRAG) systems for L'Oréal's product recommendation and description use cases. Their findings demonstrated that while Vector RAG outperformed GraphRAG slightly in terms of faithfulness and semantic similarity metrics, GraphRAG exhibited significantly faster retrieval times, approximately 1 second compared to Vector RAG's 47 seconds. This performance advantage positions GraphRAG as a potentially superior option for real-time applications where response latency is critical.

Despite these promising results, several limitations were identified in the previous implementation. The knowledge graph construction process was largely unstructured, with node and edge formation lacking systematic organization. Additionally, the free Neo4j instance used for testing did not support efficient augmentation of the existing knowledge graph with new documents, necessitating complete graph reconstruction when adding information. Perhaps most significantly, the study revealed a disparity in performance between description-based queries (where GraphRAG scored 8.9/10 in human evaluation) and recommendation-based queries (scoring 7.6/10), suggesting that the knowledge graph structure may have been better optimized for certain query types.

Our project extends the previous work by focusing specifically on knowledge graph construction methods and their impact on GraphRAG system performance. We explore various techniques for entity extraction, relationship identification, and graph structuring, evaluating how these methodologies affect both the quality and efficiency of responses. By systematically analyzing these approaches, we aim to develop guidelines for optimizing GraphRAG implementations based on specific use cases, query types, and performance requirements.

1.2 Background

GraphRAG systems represent a specialized approach to retrieval augmentation by organizing information in graph-based structures where entities serve as nodes and relationships as edges. This approach builds upon decades of knowledge representation research, from early semantic networks to modern knowledge graphs like Google's Knowledge Graph (Singhal, 2012) and Neo4j's graph database technologies. Unlike vector stores that rely primarily on similarity metrics, knowledge graphs capture explicit relationships between entities, enabling more nuanced navigation through complex information spaces.

The current landscape of GraphRAG research and implementation is rapidly evolving. Recent work by Sarmah et al. (2023) demonstrated that GraphRAG systems excel at han-

dling complex, multi-hop reasoning tasks where information must be connected across multiple entities. Yasunaga et al. (2022) showed how knowledge graphs can improve LLM reasoning through structured exploration of knowledge spaces. Meanwhile, industry implementations from companies like Microsoft have demonstrated that GraphRAG approaches particularly shine for global queries about general categories rather than specific instances (Edge et al., 2023).

However, a critical gap exists in understanding how different knowledge graph construction methodologies impact the performance of GraphRAG systems. Current approaches range from fully automated LLM-based construction (as used in the previous project with Neo4j's Knowledge Graph Builder), to rule-based extraction, to hybrid human-in-the-loop approaches. Each method presents different tradeoffs in terms of precision, recall, scalability, and domain adaptation capabilities, yet systematic evaluation of these tradeoffs remains limited.

Knowledge graph construction techniques have evolved substantially in recent years. Traditional methods relied heavily on manual curation or rule-based systems that were difficult to scale (Paulheim, 2017). More recent approaches leverage transformer models for named entity recognition and relationship extraction (Heinzerling et al., 2021). Some cutting-edge methods employ LLMs themselves to identify entities and relationships through few-shot prompting (West et al., 2022), while others use specialized graph neural networks to iteratively refine graph structures (Ji et al., 2021).

The performance of GraphRAG systems is intimately tied to the quality and structure of the underlying knowledge graph. Factors such as graph density, relationship diversity, entity granularity, and contextual embedding all influence the system's ability to retrieve relevant information efficiently. Understanding these relationships requires systematic investigation of how different construction methodologies impact downstream performance across various metrics and use cases.

1.3 Problem Statement

This project investigates how different knowledge graph construction methods influence the performance of GraphRAG systems.

While GraphRAG has shown strong potential for real-time retrieval tasks, the specific impact of graph construction methods remains poorly understood. This project addresses this gap by systematically evaluating how different construction approaches affect performances in downstream GraphRAG applications. Our goal is to identify design choices that lead to more effective and efficient GraphRAG pipelines.

1.4 Research Questions

To guide our investigation, we address the following research questions:

1. What are the existing ways to construct knowledge graphs, and how do they differ in methodology, complexity, and resource requirements?

2. How do different knowledge graph construction methods affect the retrieval accuracy, response relevance, and inference speed of GraphRAG systems?

3. What graph structures and relationship patterns are most effective for different types of queries?

By addressing these questions, our research aims to advance the understanding of GraphRAG systems and provide practical insights for their implementation in industrial applications, particularly in e-commerce and product information management contexts.

1.5 Overall Approach

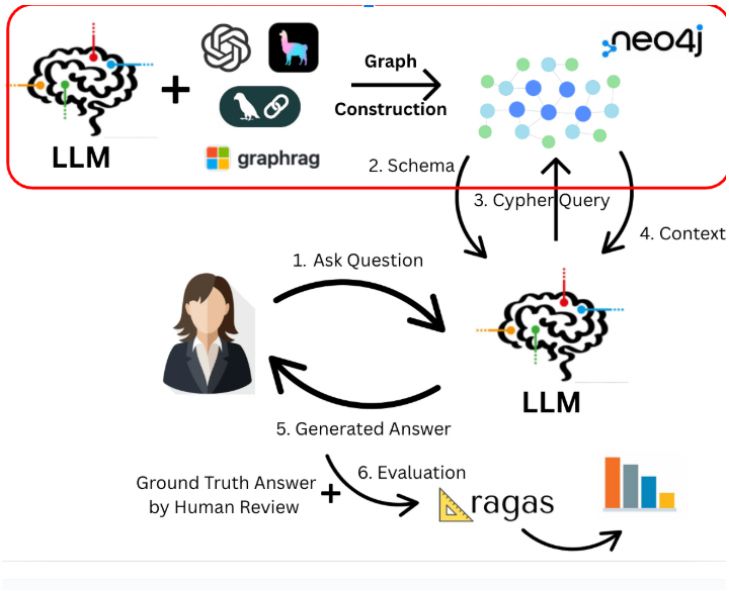

Figure 1: Approach Overview

As shown in Figure 1 , we construct a knowledge graph using LLM-driven structured extraction pipelines, each applying a schema to organize entities and relationships. The resulting graph is stored in Neo4j and queried via Cypher queries to provide contextual grounding to an LLM at inference time. Users ask natural language questions, and the LLM generates answers based on both the query and retrieved graph context. These responses are then evaluated against human-reviewed ground truths using RAGAS metrics. This end-to-end setup allows us to systematically compare the impact of graph construction methods on overall system performance.

2 Experiment Methodology

2.1 Datasets

We initially experimented with a dataset found on Hugging Face: milistu/AMAZON-Products-2023. With 117,243 rows and 15 columns, we have a wide range of products and brands. However we focused on products that are under the beauty and personal care category due to the relevancy to L'Oreal.

However, after internal discussions with our professor and mentors, we decided to find 25 products skin care products so that we can keep the dataset relevant to L'Oreal and create knowledge graphs that can be human-evaluated. This will allow us to tune knowledge graph creation methods visually.

The 25 products we chose are from `https://www.kaggle.com/datasets/kingabzpro/cosmetics-datasets`. This dataset contains 1,472 skincare products across various categories. It features comprehensive product information including brand names, product

categories, prices, customer ratings, ingredients, and skin type compatibility. Initial exploration revealed that the dataset encompasses **113** unique brands and **6** product categories: **Moisturizer, Cleanser, Treatment, Face Mask, Eye Cream, and Sun Protection**.

To create a more focused dataset for our analysis, we selected four premium brands with significant market presence: **LA MER, SK-II, ESTÉE LAUDER, and KIEHL'S SINCE 1851**. We further narrowed our scope to three essential skincare categories: **Moisturizer, Face Mask, and Eye Cream**. This refinement resulted in a subset of **73** products (36 moisturizers, 18 face masks, and 19 eye creams) from these four luxury brands.

From this refined dataset, we selected **25** signature products (**12 moisturizers, 8 face masks, and 5 eye creams**) for in-depth analysis and presentation. The selection was based on product popularity and diversity within each brand's lineup. To enhance the analytical value of the dataset, we performed several transformations:

1. **Skin Type Compatibility:** We converted the binary indicators for skin type compatibility (Combination, Dry, Normal, Oily, Sensitive) into a consolidated list format, creating a new `For Skin Type` column that explicitly lists all compatible skin types for each product.

2. **Size Information:** We manually searched and added a `Size` column containing standardized product size information (e.g., "1.7 oz (50ml)", "10 sheets", "60 capsules"), providing important context for price comparison.

3. **Product Descriptions:** We enriched the dataset with detailed product descriptions from their official websites. These descriptions offer valuable marketing context and explain the unique selling propositions of each product.

4. **Data Standardization:** We standardized brand names (replacing "É" with "E" in "ESTÉE LAUDER") and renamed columns for clarity, changing `Label` to `Product Type` and `Name` to `Product Name` to improve the intuitive understanding of the dataset structure.

The final curated dataset contains 25 skincare products across three categories and four brands, with comprehensive information including price, customer rating, ingredients, compatible skin types, product size, and detailed descriptions.

2.2 RAG

Retrieval-Augmented Generation (**RAG**) combines retrieval-based methods with generative models to improve knowledge-grounded text generation. The methodology consists of two key components:

- **Retriever:** Given a query, the retriever fetches relevant documents from a knowledge source (e.g., a vector database).

- **Generator:** The retrieved documents are then provided as context to a pre-trained generative model that generates a response based on both the input query and the retrieved knowledge.

The process follows these steps:

1. Encode the query into a vector space and retrieve the top-k relevant documents.
2. Concatenate the retrieved documents with the input query.
3. Feed the combined input into a sequence-to-sequence model to generate the final response.

RAG enhances model responses by incorporating external knowledge dynamically, reducing hallucinations compared to standard language models while maintaining flexibility for various applications, such as product recommendations and customer support in e-commerce.

In our project, **ChromaDB** is used as a vector database to store and retrieve document embeddings in a RAG pipline. Each document is embedded into a high-dimensional vector space using a pre-trained embedding model (e.g., OpenAI embeddings). This approach, referred to as **Vector RAG**, enhances the retrieval process by leveraging dense vector representations to find the most relevant documents for query augmentation.

2.3 Knowledge Graph Construction

2.3.1 Microsoft-GraphRAG

This implementation uses Microsoft's GraphRAG framework to build a flexible end-to-end pipeline that ingests unstructured text and extracts both entities and relationships to construct a knowledge graph. All of this can be done through a few simple commands in the command line. After indexing the input and constructing a knowledge graph, it is stored locally allowing for inference on a user's machine. Neo4j implementation is not directly supported but is possible to translate over to Neo4j. However, performance seems to decrease after moving the KG to Neo4j. Below are core components of our implementation:

- **Configuration Management**: All pipeline parameters live in a single `settings.yaml` file, making it very simple to swap models, enable or disable processing steps, and change storage backends without touching code.
  - `models.default_chat_model`: Chat LLM (e.g. `gpt-4o-mini`) used for graph and claim extraction
  - `models.default_embedding_model`: Text embedding model (e.g. `text-embedding-3-small`)
  - `workflows`: Ordered list of steps (e.g. `[extract_graph, extract_claims, community_reports, embed_text]`)
- **Data Ingestion and Chunking**: Documents in TXT, CSV, or JSON format are loaded using the appropriate loader and then split using overlapping chunks to preserve some context between chunks (only experimented with unstructured text files and not a CSV file).
  - `input.file_type` and `input.base_dir` guide automatic loader selection
  - `chunks.size` (e.g. 1000 chars) and `chunks.overlap` (e.g. 200 chars) tune granularity
  - CSV rows can be grouped via `chunks.group_by_columns` to avoid splitting logical records

- **Knowledge Graph Extraction Workflow**: The `extract_graph` module guides the LLM through multiple "gleaning" passes to identify entities and relations defined in the schema. It then prunes results into a coherent graph. Gleanings are iterative passes through data to refine extraction.
  - `entity_types`: e.g. {`Product Type`, `Brand`, `Ingredient`, `Price`}
  - `relationship_types`: e.g. {`FROM_BRAND`, `CONTAINS_INGREDIENT`, `HAS_PRICE`}
  - `max_gleanings`: Number of iterative passes (e.g. 3) to refine extraction
  - Pruning merges duplicates and filters low-confidence edges before persistence
- **Query-Time Retrieval Strategies**: At inference time, MS-Graph RAG utilizes its flexibility by combining structured graph traversal and semantic vector search. MS-Graph RAG has options for local search and global search, allowing for further flexibility.
  - `local_search`: Expands from one or more seed entities up to a configurable hop count
  - `global_search`: Uses precomputed community summaries in a map–reduce style to quickly narrow context
  - `drift_search`: Blends local graph context with vector similarity over community reports for dynamic follow-up questions

2.3.2 Llama Index

This implementation constructs a symbolic knowledge graph from structured tabular data using LlamaIndex's graph abstraction framework. The original implementation is designed for unstructured articles, which we adapted to our structured CSV input. The resulting graph is stored in Neo4j and enriched with document-level embeddings to enable hybrid retrieval. The pipeline consists several main components:

- **Data Ingestion and Preprocessing:** The pipeline begins by ingesting structured data using `pandas`. Each row is converted into a `Document` object using the LlamaIndex library, combining product attributes into a unified text block and storing selected fields (e.g., `title`, `description`) as metadata. These document objects serve as the core input unit for both embedding and graph construction workflows.
- **Entity and Relationship Extraction:** Structured knowledge is extracted using a system-level prompt that converts each document into a collection of entities labeled with predefined types, and their pairwise relationships. The prompt follows a few-shot format and explicitly specifies a JSON schema with two top-level keys: `entities` and `relationships`, guiding the model toward consistent output.

  Each entity includes:
  - `entity_name`: the canonical name of the entity
  - `entity_type`: a type drawn from a closed schema (e.g., Product, Brand, Ingredient)
  - `entity_description`: a concise description grounded in the document content

Relationships are defined similarly, linking known entities with a typed label (e.g., CONTAINS, SUITABLE_FOR) and a brief explanation.

To operationalize this, we use LlamaIndex's `GraphRAGExtractor`, which invokes the language model with the defined prompt and parses the JSON output using a custom function. Extracted triples are converted into structured nodes and edges enriched with metadata for downstream use in the knowledge graph.

- **Graph Construction and Storage:** The extracted entities and relationships are assembled into a property graph using a custom extension of LlamaIndex's `Neo4jPropertyGraphStore`. Each entity and relationship is enriched with descriptive metadata generated by the LLM during extraction. The graph is stored in Neo4j. The graph construction step connects document-derived `Chunk` nodes with inferred entities to enable traceability. The internal graph structure is later converted to a NetworkX graph for community detection.

- **Community Construction and Summarization:** Using the internal representation of the graph, we apply the hierarchical Leiden algorithm to cluster closely related nodes into communities. For each community, the system synthesizes a natural language summary by prompting the LLM with a structured list of relationship triples (entity1, entity2, relation). These summaries capture the key relational semantics of each subgraph and are cached for later use.

- **Community-Aware Query Engine:** The pipeline includes a custom `GraphRAGQueryEngine`, which routes user queries through the community summaries produced by `GraphRAGStore`. At inference time:

  1. The query engine retrieves all available community summaries.
  2. For each summary, the system uses the LLM to generate a candidate answer based on the user query and the corresponding summary.
  3. The individual answers are passed back to the LLM to be aggregated into a single coherent response.

2.3.3 OpenAI Pipeline

This implementation uses OpenAI's GPT model to extract entities and relationships from structured product data and constructs a knowledge graph stored in Neo4j. The pipeline processes CSV inputs using templated prompts and parses structured JSON outputs to generate Cypher graph statements.

- **Prompt-Based Extraction Using GPT:** For each structured product row, a prompt is dynamically constructed and sent to the OpenAI GPT API. The model is instructed to return a JSON object containing two keys: `entities` and `relationships`. Each entity includes a `label`, `id`, and relevant attributes such as `name`, `brand`, or `price`. Relationships are defined as pipe-delimited strings of the form:

  `"source_id|RELATION_TYPE|target_id"`.

  The prompt supports open-ended schema extraction, allowing the model to define entity types, relationship types, and attribute sets without relying on a predefined ontology. This offers high flexibility and adaptability to new domains, since no hard-coded schema is required. However, this also reduces semantic control: the model

often generates structurally valid but semantically weak relationships. Because relationship extraction depends entirely on the prompt, the system may produce abundant but low-relevance outputs in the absence of stricter filtering.

Like other approaches, this method includes a few-shot example in the prompt that demonstrates the expected JSON structure and shows how to normalize entity IDs and format labels correctly. This helps guide the model to produce consistent, parseable output.

- **Output Parsing and Validation:** The raw output from the model is processed using regular expressions to extract the JSON block. Any responses that fail to conform to the expected structure are skipped and logged. Successfully parsed outputs are passed to the next stage for graph construction.

- **Cypher Statement Generation and Neo4j Storage:** For each extracted entity and relationship, Cypher `MERGE` statements are generated. These statements ensure that the nodes are uniquely created based on `id` values and that all relevant properties are included. Relationships are added only if both source and target entities exist in the parsed data. The generated Cypher statements are saved to disk and can be executed directly against a Neo4j instance using the official driver.

2.3.4 LLM Graph Transformer

This implementation utilizes LangChain's `LLMGraphTransformer` framework, which allows for the systematic extraction of entities and relationships from textual data. The core of our implementation is built on the following components:

- **LLM Integration**: The framework is designed with flexibility to adapt to various LLM providers. While our implementation uses OpenAI's GPT-4 model through the ChatOpenAI interface, the architecture allows for easy substitution with alternative LLM services or local models.

- **Schema Definition**: We define a structured schema specifying allowable node types and relationship types. Our knowledge graph schema includes:
  - Node types such as `Brand`, `Product_type`, `Product`, `Price`, `Rating`, `Skin_type`, `Size`, `Description`, and `Ingredients`
  - Relationship types including `FROM_BRAND`, `HAS_PRICE`, `HAS_RATING`, `FOR_SKIN_TYPE`, `IN_SIZE`, `HAS_DESCRIPTION`, `HAS_PRODUCT_TYPE`, and `CONTAINS_INGREDIENT`

- **Document Processing Pipeline**: We process CSV data containing cosmetic product information through the following steps:
  - Loading documents using `CSVLoader`
  - Splitting documents into manageable chunks with `CharacterTextSplitter`
  - Computing embeddings for each document using OpenAI's embedding model
  - Storing both documents and their embeddings in a Neo4j graph database

- **Graph Construction**: For each document processed:
  - We embed the document content

  - Create a Document node and a Chunk node in the graph
  - Use the `LLMGraphTransformer` to extract entities and relationships
  - Connect the extracted entities to the document structure
  - Add the graph documents to the Neo4j database

- **Vector Indexing**: We create a vector index on the document embeddings to enable efficient similarity search during retrieval.

This approach allow us to either use the default `LLMGraphTransformer` functionality or specify custom node and relationship types according to the domain requirements. When operating in strict mode, the transformer ensures that only the specified node and relationship types are created, maintaining the integrity of the knowledge graph structure.

2.4 Graph RAG

Graph RAG extends the Vector RAG framework by incorporating structured knowledge graphs into the retrieval process. This approach enhances the model's ability to generate contextually relevant and semantically rich responses leveraging explicitly defined relationships between entities.

For example, consider two entities **Apple AirPods Pro** and **Apple AirPods Pro Case** in the **Amazon Product Dataset**. They might be connected through a relationship, such as:

Apple AirPods Pro Case —[**ACCESSORY FOR**]— Apple AirPods Pro

At the implementation level, Graph RAG enhances the retrieval and generation pipeline by integrating a graph database, such as **Neo4j**. The system typically consists of three core components:

1. **Retriever**: Traverses the knowledge graph to identify relevant nodes and edges, leveraging structured relationships to improve retrieval accuracy.

2. **Ranker**: Prioritizes retrieved subgraphs based on contextual relevance, ensuring the most informative and meaningful knowledge is passed to the model.

3. **Generator**: Conditions the LLM on both graph-based insights and retrieved text passages, allowing for responses that are both contextually rich and factually grounded.

Neo4j's Cypher queries allow efficient extraction of interconnected concepts, making Graph RAG particularly effective for domains requiring reasoning over structured knowledge.

2.5 Evaluation Metrics

We evaluate the performance of GraphRAG using the `ragas` library, which provides a suite of metrics designed to assess the quality of responses in retrieval-augmented generation (RAG) systems. The key evaluation criteria include **Response Relevancy**, **Faithfulness**, and **Semantic Similarity**. Each metric captures a different aspect of response quality, as detailed below.

2.5.1 Response Relevancy

The *Response Relevancy* metric measures how closely the generated response aligns with the user input. It uses cosine similarity to compare the user input embedding with embeddings of artificial questions generated from the response. The procedure is as follows:

1. Generate a set of synthetic questions (typically 3) based on the response.
2. Compute the cosine similarity between the embedding of the user input (E_o) and each question embedding (E_{g_i}).
3. Average the cosine similarity scores to compute the final relevancy score:

$$\text{Answer Relevancy} = \frac{1}{N} \sum_{i=1}^{N} \frac{E_{g_i} \cdot E_o}{\|E_{g_i}\| \|E_o\|}$$

2.5.2 Faithfulness

The *Faithfulness* metric evaluates the factual consistency of the response with respect to the retrieved context. A response is considered faithful if its claims can be inferred from the provided context. The score is computed as:

$$\text{Faithfulness Score} = \frac{\text{Number of claims supported by the context}}{\text{Total number of claims in the response}}$$

This involves:

- Extracting claims from the response.
- Validating each claim against the retrieved context.

2.5.3 Semantic Similarity

Semantic Similarity quantifies the degree to which the generated response semantically aligns with a ground truth answer. This metric reflects conceptual accuracy rather than surface-level similarity, and is computed by comparing the embeddings of the response and the reference answer. A higher score indicates greater semantic overlap.

By employing these metrics through the `ragas` library, we obtain a comprehensive evaluation of GraphRAG's ability to produce relevant, factual, and semantically accurate responses.

3 Results

3.1 Evaluation Results

Figure 2 presents the comparative evaluation of various GraphRAG-based methods and a VectorRAG baseline using the `ragas` metrics: *faithfulness*, *answer relevancy*, and

semantic similarity. Among all GraphRAG methods, Microsoft GraphRAG and LLMGraphTransformer consistently demonstrates strong performance across all metrics, particularly achieving the highest scores in *faithfulness* and *answer relevancy*. In contrast, while Vector RAG performs relatively well in *answer relavancy*, it shows a significant drop in *faithfulness*, indicating potential hallucinations or unsupported claims in its responses. Notably, Microsoft GraphRAG exhibits a substantial improvement over its generic counterpart, especially in *faithfulness*, demonstrating the benefit of domain-specific tuning. Methods such as LLMGraphTransformer and LlamaIndex API show competitive performance, but with more variation in *answer relevancy*. Overall, these results highlight the effectiveness of combining graph-based retrieval like GraphRAG and LLMGraphTransformer with LLMs and emphasize the importance of fine-tuning and method design in achieving high-quality RAG outputs.

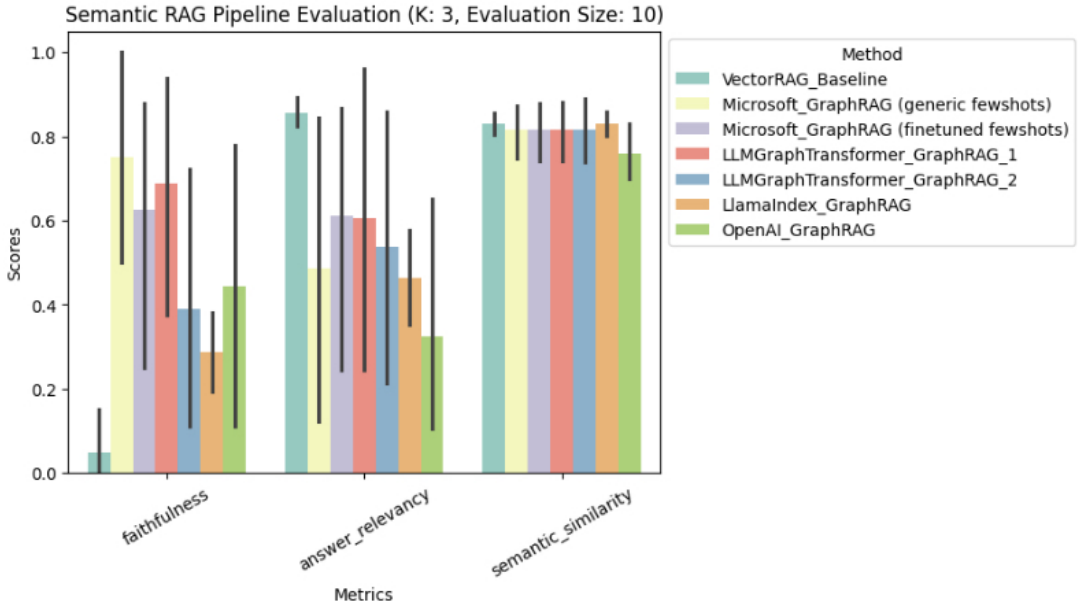

Figure 2: RAGAS Evaluation of Vector RAG & Graph RAG

3.2 Conclusions

- For GraphRAG, Microsoft GraphRAG and LLMTransformer are better APIs to construct knowledge graph
- Knowledge graph construction will lead to information loss of well-structured data (e.g. csv file)
- Few-shot learning is a good technique for context enhancement As the core component of pipeline, LLM should have strong information processing capability and cypher query coding ability to generate reasonable answers

4 Future work

Our explorations this semester on knowledge graph construction and GraphRAG implementation have revealed several directions for further research. The following areas

represent crucial next steps to enhance the effectiveness, efficiency, and scalability of our approach.

4.1 Knowledge Graph Construction Optimization

4.1.1 LLM-Driven Auto-tuning Process

Microsoft proposed to develop a comprehensive auto-tuning pipeline that begins with providing text samples to LLMs to identify domain characteristics and create appropriate personas (Fernández, et al., 2024). This process would continue with the generation of domain-specific prompts containing precise graph extraction examples based directly on the data. These prompts would undergo systematic refinement through iterative testing before being applied to the GraphRAG implementation. This approach would likely enhance domain specificity and extraction quality while reducing the need for manual prompt engineering.

4.1.2 Graph Refinement Techniques

Future work should explore post-construction refinement processes to eliminate redundant nodes and strengthen meaningful connections within the knowledge graph. This includes using LLM to make edits on the existing graph structures, applying rule-based techniques to refine the graph, and using RL pipelines for graph refinement.

4.1.3 Hierarchical Structure Implementation

Incorporating hierarchical elements into our knowledge graph represents a significant opportunity for improvement. We used Microsoft's proposed element summaries and community reports methodologies to capture multi-level relationships among beauty products. However, this is an automated processing in their graphrag package. They used Leiden algorithms to perform community detection within the graph. It is possible that such algorithm could be replaced by other type of graph algorithms to improving performance.

4.2 Robust Cypher Query Generation

Our research revealed that large language models struggle to generate functioning Cypher queries based solely on their knowledge and provided graph schema, with performance remaining suboptimal even when few-shot examples were provided. This caused the information retrieval from the knowledge graph to be inaccurate. Therefore, it may be more effective to implement an independent agent specifically designed for domain-specific Cypher query generation rather than relying on general-purpose language models.

4.3 Knowledge Graph Scalability and Maintenance

4.3.1 Scaling to Industrial Implementation

Our project deliberately utilized a small dataset of only 25 products to facilitate clearer exploration and visibility during the research phase. This controlled scope enabled us to effectively identify and compare different KG construction methods while maintaining manageable processing times. The limited dataset size allowed for detailed

analysis of relationship structures and query performance across different implementation approaches.

Industrial applications would require the knowledge graph to handle datasets 100 or even 1000 times larger than our experimental implementation. Future work must explore efficient construction and storage techniques for these significantly larger graphs. This includes investigating optimized indexing structures, partitioning strategies, and potentially distributed storage solutions that maintain query performance at scale. Research should particularly focus on addressing the computational bottlenecks we observed when using the OpenAI API for larger datasets, potentially through specialized batching strategies or alternative extraction approaches.

4.3.2 Incremental Update Mechanisms

A critical limitation of our current implementation is the one-time construction of the knowledge graph without provisions for updates. Production environments require efficient mechanisms to integrate new product information without complete graph reconstruction. Future work should develop change detection systems that identify only the portions requiring updates, alongside reconciliation algorithms that preserve existing relationships while integrating new information. These capabilities are essential for maintaining graph relevance in dynamic e-commerce environments where product catalogs frequently change.

4.4 Evaluation Framework Enhancement

To properly assess system effectiveness, it maybe better to develop specialized metrics that evaluate answer quality specifically for beauty products. Future evaluation frameworks should consider factors like product compatibility, ingredient interactions, and alignment with user preferences. These metrics would provide more meaningful performance indicators than general-purpose measures and offer clearer guidance for optimization efforts targeting the specific needs of L'Oreal's customer base.

Future work should include systematic comparison of different graph construction methodologies across standardized query sets. This analysis would benchmark performance differences between filtering and semantic query handling to identify specific optimization opportunities. By clearly understanding the strengths and limitations of each approach relative to different query types, we can develop targeted improvements addressing the unique challenges of both product description and recommendation tasks.

5 Ethical Considerations

This project utilizes publicly available cosmetics datasets containing only factual product information such as brand names, ingredients, and prices, which poses minimal ethical concerns. No personally identifiable information or sensitive customer data is involved in the knowledge graph construction or evaluation processes. Since the implementation is intended for L'Oréal's internal product information systems using their own data, it operates within the company's existing data governance frameworks and ethical standards.

6 Contributions

- **Brandon Keung:** Team Captain, Neo4j Research & Implementation, KG Construction through OpenAI Pipeline, KG Construction through Microsoft-GraphRAG, Report Writing
- **Jiaheng Dai:** Vector RAG Pipeline, OpenAI Pipeline, Llama Index Approach, Report Writing
- **Jiayi Dong:** Knowledge Graph Construction & LangChain API Research, KG construction, Report Writing
- **Pei Tian:** Evaluation Pipeline, Semantic GraphRAG retrieval, Vector RAG retrieval, Report Writing

7 GitHub

`https://github.com/engie4800/dsi-capstone-spring-2025-Loreal-graph-creation/tree/main`

8 Reference

[1] Lewis, Patrick, et al. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401, arXiv, 12 Apr. 2021. arXiv.org, https://doi.org/10.48550/arXiv.2005.11401.

[2] Singhal, A. (2012, May 16). Introducing the knowledge graph: Things, not strings. Google. https://blog.google/products/search/introducing-knowledge-graph-things-not/

[3] Sarmah, Bhaskarjit, et al. HybridRAG: Integrating Knowledge Graphs and Vector Retrieval Augmented Generation for Efficient Information Extraction. arXiv:2408.04948, arXiv, 9 Aug. 2024. arXiv.org, https://doi.org/10.48550/arXiv.2408.04948.

[4] Yasunaga, Michihiro, et al. QA-GNN: Reasoning with Language Models and Knowledge Graphs for Question Answering. arXiv:2104.06378, arXiv, 13 Dec. 2022. arXiv.org, https://doi.org/10.48550/arXiv.2104.06378.

[5] Edge, Darren, et al. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. arXiv:2404.16130, arXiv, 19 Feb. 2025. arXiv.org, https://doi.org/10.48550/arXiv.2404.16130.

[6] Paulheim, H. (2017). Knowledge graph refinement: A survey of approaches and evaluation methods. Semantic Web, 8(3), 489-508.

[7] Heinzerling, Benjamin, and Kentaro Inui. Language Models as Knowledge Bases: On Entity Representations, Storage Capacity, and Paraphrased Queries. arXiv:2008.09036, arXiv, 21 Apr. 2021. arXiv.org, https://doi.org/10.48550/arXiv.2008.09036.

[8] West, Peter, et al. Symbolic Knowledge Distillation: From General Language Models to Commonsense Models. arXiv:2110.07178, arXiv, 28 Nov. 2022. arXiv.org, https://doi.org/10.48550/arXiv.2110.07178.

[9] Ji, Shaoxiong, et al. A Survey on Knowledge Graphs: Representation, Acquisition and Applications. arXiv:2002.00388, arXiv, 1 Apr. 2021. arXiv.org, https://doi.org/10.48550/arXiv.2002.00388.

[10] Fernández, A. G., Smith, K., Bradley, J., Edge, D., Trinh, H., Smith, S., Cutler, B., Truitt, S., & Larson, J. (2024, November 5). Graphrag Auto-tuning provides rapid adaptation to new domains. Microsoft Research. https://www.microsoft.com/en-us/research/blog/graphrag-auto-tuning-provides-rapid-adaptation-to-new-domains/

[11] LlamaIndex Contributors, LlamaIndex: Interface between LLMs and your data. GitHub, 2024. [Online]. Available: https://github.com/run-llama/llama_index. [Accessed: May 17, 2024].

[12] D. Wang, J. Liu, S. Jain, E. Koshy, E. Ou, J. Tan, S. Sanyal, M. Mahadevan, S. Banerjee, E. W. Han et al., "LlamaIndex: A Data Framework for Building Context-Augmented LLM Applications," arXiv preprint arXiv:2404.16130, 2024. [Online]. Available: https://arxiv.org/abs/2404.16130