POLITECNICO DI TORINO

Master's Degree in Data Science and Engineering

Master's Degree Thesis

Structured Retrieval-Augmented Generation for Enterprise Knowledge Management

Supervisors

Prof. ANDREA BOTTINO

Ing. MARIAGRAZIA CARDILE

Ing. GIULIO NENNA

Candidate

UMBERTO PICCARDI

DECEMBER 2025

Abstract

This thesis addresses the problem of onboarding and knowledge retrieval in modern companies, where documentation is often voluminous, generic and fragmented across many systems. Retrieval-augmented generation (RAG) language models combine a search step with text generation: the system retrieves relevant passages from knowledge bases and feeds them to the model to produce more up-to-date and accurate responses. However, traditional RAG systems, based on simple vector or lexical search, struggle with complex questions that require linking information from different domains and synthesising it in a coherent manner.

We suggest a RAG framework for industrial settings that combines a structured retrieval approach with a knowledge graph in order to overcome these drawbacks. Explicit relationships between concepts and entities are added to traditional retrieval by the graph-based design, which enables the system to reason across related data and produce more logical, context-aware responses. This method improves overall factual consistency and explainability while strengthening RAG's capacity to manage intricate, cross-domain inquiries.

The RAGAs framework, a collection of LLM-based metrics intended to evaluate retrieval and generation quality, is the foundation of the evaluation. Using a standard open-domain dataset, we compared the effectiveness of the graph-augmented approach against a baseline RAG in terms of faithfulness, answer relevancy, and context precision. The results provide an initial validation of the framework's potential before its application to enterprise documentation environments.

Overall, this thesis contributes: (i) an analysis of the challenges posed by onboarding and fragmented enterprise knowledge, (ii) a graph-augmented RAG framework based on community summaries and local/global retrieval, and (iii) a holistic evaluation demonstrating the benefits of graph-based retrieval for enterprise knowledge management.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large Language Models (LLMs) have ushered in a new era of AI capabilities, demonstrating an unprecedented ability to generate human-like text and reason about complex questions. In industry, this has sparked widespread interest in deploying LLM-powered assistants and tools that can leverage an organisation's internal knowledge. However, a key limitation of standalone LLMs is their tendency to produce confident but unsupported answers when asked about details outside their trained knowledge. To address this, the paradigm of *retrieval-augmented generation* (RAG) has emerged as a popular solution. RAG techniques marry LLMs with information retrieval: the system fetches relevant documents or data from external sources and provides them as context to the LLM, which then generates a grounded answer. This approach allows LLMs to access private or domain-specific data on the fly and significantly curbs unsupported claims. As a result, RAG has been widely adopted in real-world applications—from enterprise assistants that answer questions about policies and procedures to intelligent search systems that tailor responses using proprietary data. In essence, RAG offers a promising way to inject factual grounding into generative AI, enabling large models to serve as helpful assistants within the context of an organisation's knowledge.

1.1 Challenges of Deploying RAG in the Enterprise: Overload, Fragmentation and Silos

Despite these advances, deploying RAG systems in complex enterprise environments is far from trivial. New employees onboarding onto large projects often encounter a perfect storm of information challenges. First, documentation is frequently *overly broad and overwhelming*. Organisations tend to accumulate extensive documents, wiki pages, runbooks and design manuals that attempt to cover everything for everyone. Yet no single artefact meets the diverse needs of all stakeholder groups

equally. Highly detailed reports may span tens of pages—useful to experts, but inaccessible to newcomers or those in different roles. A frontend engineer joining a project, for instance, might receive a monolithic handbook covering frontend, backend, data engineering and more, without any mechanism to filter content by role or expertise. The lack of role-specific filtering means that relevant details are buried among irrelevant information, making it arduous for an individual to distil what matters for *their* work. The result is not only frustration, but also wasted time: employees spend a non-trivial portion of their day searching for and gathering information instead of applying it.

Secondly, enterprise knowledge is often *fragmented across many sources*. In a typical large project, information may be scattered in a constellation of disconnected systems: requirements in wiki spaces, API docs in Markdown files, architectural decisions in ticketing systems and data definitions across heterogeneous databases (for example, Oracle, PostgreSQL and BigQuery). Different teams and departments maintain separate knowledge bases, leading to classic knowledge silos. This fragmentation forces employees to manually navigate multiple systems to piece together answers. It resembles an organisational amnesia in which the left hand does not know what the right hand learned yesterday. The silos are exacerbated by the division of development environments: commonly there are separate databases and documentation for development, system/integration and production. A new hire may find that instructions for the development environment differ from production troubleshooting guides, with no unified view bridging these. This heterogeneity makes it extremely challenging to ask cross-cutting questions such as, "How does the front-end error logging in the test environment correlate with known backend issues in production?" Traditional portals or search tools are not equipped to aggregate such multifaceted knowledge in a coherent way.

Given these challenges, there is a clear need for a smarter RAG system that can *help employees navigate complex enterprise documentation in a more targeted and intelligent way*. The goal is to transform the onboarding and knowledge retrieval experience from a disjointed scavenger hunt into a cohesive, context-aware dialogue. Such a system should allow *cross-functional questions*—queries that span multiple domains or teams—and provide answers that reflect a unified perspective across all relevant sources and environments. For example, an engineer might ask: "What are the main differences between the legacy data pipeline and the new one and how do they affect front-end load times?" Answering this requires pulling together information about data engineering (pipelines), backend (performance impacts) and frontend (load times), likely from different repositories. A sophisticated RAG system would interpret this complex question, retrieve pieces from each silo (perhaps a design note from the data team, a performance report from operations and a front-end architecture document) and *compose an integrated answer*. In essence, employees need an intelligent knowledge concierge—one that can screen

through heterogeneous databases, bridge knowledge silos and tailor information to the user's role and context. This motivates the work in this thesis: designing a retrieval-augmented generation framework that is aligned with the *multi-source, multi-environment reality* of modern enterprises.

1.2 From Vector Matches to Relational Understanding

Recent developments in RAG research point towards a promising solution: *GraphRAG*. GraphRAG augments the traditional RAG architecture with knowledge graphs, bringing explicit structure and relationships into the retrieval process. In conventional RAG, the retriever typically uses vector similarity to find texts semantically similar to the query and the generator then produces an answer from those texts. This works well for many straightforward queries but often falters on questions that require *connecting disparate pieces of information* or performing multi-hop reasoning. For instance, Answering a question such as "Who authored the design document for the system that replaced the legacy database and what were the main improvements?" requires establishing a sequence of facts: identifying the replacement system, locating its design document, and then determining the author of that document and summarising the improvements. A vanilla RAG system may struggle because relevant information is spread across different documents and not explicitly linked. In contrast, GraphRAG introduces an *entity-centric and relational view of knowledge*: it represents information as a graph of nodes (*entities*) and edges (*relationships*) rather than as isolated text chunks. By doing so, GraphRAG can explicitly capture how pieces of data relate to each other. For example, in the case of a company document which system replaced which, who authored a document and which component belongs to which project, instead of relying solely on implicit semantic similarity. This structured, graph-based organisation is inherently well-suited for complex queries, because it preserves relational context and supports principled traversal.

Operationally, GraphRAG leverages a pipeline of query-focused summarisation, global reasoning and hierarchical, entity-based responses to overcome limitations of traditional RAG. In the indexing stage, raw documents are processed into a knowledge graph. Key facts or claims, such as people, projects or systems, as well as the relationships between relevant entities are extracted from the text. This yields an initial graph of nodes and edges representing the knowledge domain. Next, a community detection method groups the graph's nodes into *communities of related entities*. Each community might represent, for example, a cluster of documents and entities related to a specific project or a functional area. These communities

are organised hierarchically (larger communities may consist of several smaller sub-communities), reflecting different granularities of the knowledge space. Crucially, GraphRAG then generates a summary for each community using a bottom-up approach. The summary encapsulates the main entities in that community, their relationships and the central themes. The outcome of indexing is therefore not just a graph, but a graph with a structured "memory": a set of summaries that provide an overview of each cluster of knowledge. This is invaluable for query answering, because it gives the system a "map" of the knowledge landscape which can be consulted to answer broad or cross-cutting questions.

At query time, GraphRAG can operate in different modes to best utilise this structure. For *holistic or broad questions* that span multiple topics, a *global search* strategy leverages the pre-generated community summaries as context. The system identifies which knowledge communities are relevant to the query and pulls their summaries or portions of them to feed into the language model. Because these summaries were designed to highlight key relationships and facts, they provide a well-informed starting point for the model to reason globally about the data. For *specific or entity-focused questions*, a *local search* mode is employed. The system finds the particular entity or entities mentioned in the query within the graph, then retrieves not only documents directly about that entity but also *neighbours* in the graph—related entities and their associated information. For example, if the query asks about a particular microservice, the local search could retrieve data on that microservice and connected nodes such as its owning team, upstream and downstream systems and recent incidents, assembling a rich context for the model. By spreading out along the graph's edges, GraphRAG ensures that the model receives pertinent facts that are one or two hops away from the query focus, which a vanilla RAG might miss if those facts are embedded in separate documents. Because the knowledge graph provides an organising principle, responses can be presented in a hierarchical, entity-based manner. If a query asks for a comparison between two systems, the answer can enumerate points under each system (entity) or by aspect, mirroring the structure of the underlying data.

By structuring knowledge as communities of related entities and aggregating information through graph-based summaries, GraphRAG directly tackles several shortcomings of traditional RAG. A common complaint of baseline implementations is brittleness with complex queries: the retriever may bring back documents that each only partially address the question, leaving the model to guess connections—or worse, miss them entirely. GraphRAG's knowledge graph allows the system to *synthesise* insights that were never explicitly written down in one place. It breaks down walls between silos: if answer-relevant facts live in separate sources, the graph's relationships and community summaries bring them together. In practice, this structured approach improves completeness and faithfulness of answers, reduces the chance of unsupported claims and provides a richer foundation for reasoning.

Another key advantage is explainability and support for *global reasoning*, which becomes even more critical when AI systems move beyond simple question answering into the realm of *AI agents*. Organisations are increasingly exploring autonomous or semi-autonomous agents—systems that can not only answer questions but also perform multi-step tasks, make decisions and interact with tools or services. These agents maintain a conversation or work on a problem over an extended period and thus require durable memory and auditable reasoning. Here, the limitations of standard RAG become apparent: a naive retrieve-then-generate loop over isolated text snippets is often insufficient for complex, multi-step tasks. GraphRAG offers a compelling synergy with AI agents by serving as a *structured knowledge substrate* for agent reasoning. Because GraphRAG's retrieval is relational and can retrieve entire subgraphs of related information, an agent can use it to maintain context across multiple steps. Moreover, the structured nature of the graph means that an agent's thought process can be traced via the nodes and edges followed. This level of auditability and transparency is valuable for building trust, ensuring compliance, and debugging.

A critical aspect of developing any RAG system, especially one that claims improved reasoning, is *evaluation*. Traditional NLP metrics such as BLEU or ROUGE are poorly matched to the goals of RAG because they measure surface similarity rather than evidence use, factuality, or retrieval quality. We therefore adopt an automated, RAG-specific evaluation framework that examines a set of key questions: whether the generated answer remains faithful to the retrieved sources (*faithfulness*), whether it directly addresses the user query (*answer relevancy*) and whether the retrieved context is both focused and sufficiently comprehensive (*context precision* and *context recall*). These dimensions collectively paint a more faithful picture of a RAG system's performance, especially for reasoning-centric outputs. They allow us to quantify improvements—e.g., whether GraphRAG provides more faithful answers or exhibits higher context recall on multi-hop questions than a baseline dense or sparse system. While there is not yet a universally accepted standard for evaluating the reasoning quality of RAG outputs, using such a framework enables systematic, reproducible comparisons and highlights where graph-based retrieval offers tangible benefits.

Scope and Objectives

This thesis introduces and investigates *GraphRAG* as an advanced RAG system intended to support cross-functional question answering over complex knowledge. We compare GraphRAG with more traditional approaches, including a purely dense-vector pipeline, a purely sparse (keyword-based) pipeline, and a hybrid combination. We examine how each of these approaches fares when it comes to

connecting dispersed pieces of information.

To enable objective comparison, we evaluate these systems using an automated metric suite focused on faithfulness, answer relevancy and context precision/recall. Concretely, experiments are conducted on the **WikiEval** dataset (`explodinggradients/WikiEval`, split `train`), which comprises a diverse set of Wikipedia pages and associated queries and on a subset of the **hotpotQA** dataset `hotpotqa/hotpot_qa`, a large-scale Wikipedia-based multi-hop QA benchmark with questions requiring reasoning over multiple supporting documents. This choice provides a controlled, non-proprietary testbed for stress-testing retrieval and reasoning without exposure to sensitive data.

While the experimental study uses public data, the broader motivation is to inform a subsequent deployment phase in enterprise settings, such as making use of internal repositories, including the structure of different databases and environments. That translation to a production context, along with potential extensions, is discussed in the future work rather than treated as a contribution of the present experiments.

Thesis Structure and Contributions

The remainder of the thesis is organised as follows:

- **Chapter 2 — Background and Related Work:** Large Language Models, the RAG paradigm and variants and graph-based retrieval/summarisation.
- **Chapter 3 — System Design (GraphRAG):** Data processing, knowledge graph construction, community detection, hierarchical summarisation and query workflows (global/local).
- **Chapter 4 — Dataset and metrics:** Datasets (*WikiEval*, *hotpotQA*) and the automated metrics used for assessment.
- **Chapter 5 — Results and Discussion:** Quantitative results across metrics and query types; comparative analysis of strengths and limitations.
- **Chapter 6 — Conclusions and Future Directions:** Summary of findings, limitations and prospective extensions, like application for enterprise data sources and, where appropriate, agentic orchestration as a longer-term direction.

Contributions. In summary, the main contributions of this thesis are:

- **Problem framing and requirements.** We articulate challenges in cross-functional knowledge access across heterogeneous sources and environments

and derive requirements for a RAG system capable of connecting dispersed information.

- **Design and implementation of GraphRAG.** We present a practical GraphRAG architecture that incorporates knowledge graph construction, community detection and hierarchical summarisation to enable query-focused reasoning.
- **Comprehensive evaluation on a public corpus.** We empirically compare GraphRAG with dense, sparse and hybrid baselines on the *WikiEval* and *hotpotQA* datasets using an automated metric suite focused on faithfulness, answer relevancy and context precision/recall.
- **Methodological insight.** By applying and analysing RAG-specific metrics, we provide insight into how reasoning-centric RAG systems can be measured, identifying gaps and suggesting areas where evaluation standards could be strengthened.

Overall, the thesis shows that a GraphRAG system can improve the completeness and faithfulness of answers on complex queries in a controlled public setting, laying the groundwork for future application to enterprise knowledge with heterogeneous data sources. Future directions include transferring the approach to internal repositories, which could be database schemas and environment-specific assets, and, where appropriate, exploring orchestration patterns that further support multi-step reasoning in applied contexts.

Chapter 2

Related Work

2.1 Large Language Models (LLMs)

Large Language Models (LLMs) are neural networks with very large parameter counts trained on massive text corpora to model and generate human language. Recent advances, in particular Transformer-based models, have led to substantial gains on many NLP benchmarks. These models can approximate human-level performance on diverse tasks. For example, Brown et al. [1] trained GPT-3, an autoregressive Transformer-based LM with 175 billion parameters and achieved strong performance in a purely few-shot setting on translation, question-answering, cloze completion and even on-the-fly reasoning tasks, without any task-specific fine-tuning. LLMs have become central to modern NLP because they pre-train on large corpora and can then be adapted, via prompting or fine-tuning, to tasks such as summarization, translation, dialogue, coding, etc., often delivering state-of-the-art results. In short, ever-larger models trained with Transformer architectures have driven a revolution in language generation and understanding.

2.1.1 Transformer Architecture

The core architecture underpinning most LLMs is the Transformer[2]. A Transformer layer, in both encoder and decoder, has two main sub-layers: a multi-head self-attention mechanism and a position-wise feed-forward network. Each sub-layer is wrapped with a residual -"add & norm"- connection and layer normalization. The decoder adds a third sub-layer of encoder–decoder attention, attending to the encoder's outputs. Because Transformers have no recurrence or convolution, they use positional encodings, added to input embeddings, to incorporate sequence order.

The building blocks of a Transformer layer can be summarised as:

- **Positional Encoding:** Since the model is permutation-invariant, fixed positional encodings, such as sinusoidal functions of token position, are added to input embeddings. These encodings have the same dimension as the embeddings and let the model take token order into account.

- **Multi-head Self-Attention:** Each token attends to all tokens in the (input or decoder) sequence. Scaled dot-product attention computes a weighted sum of value vectors, where weights come from query-key compatibilities. In multi-head attention, this process is done in parallel across several "heads"

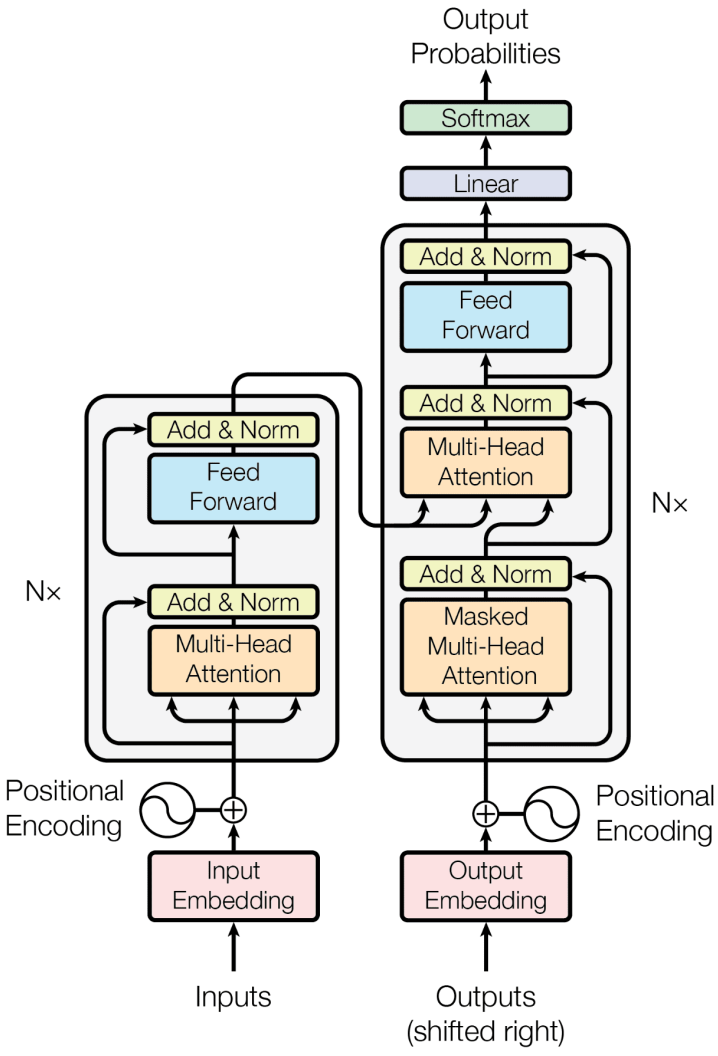

Figure 2.1: Tranformer architecture.

with different learned projections, allowing the model to capture information from different representation subspaces.

- **Feed-Forward Network:** A simple fully-connected network applied independently at each position. Typically it has two linear layers with a ReLU activation in between. It transforms each token's representation to an intermediate space and back, allowing per-position nonlinearity.
- **Add & Norm (Residual Connection + LayerNorm):** After each sub-layer, the input is added to the sub-layer output (residual connection) and the sum is normalized. In formula form:

$$\text{output} = \text{LayerNorm}(x + \text{Sublayer}(x)) \tag{2.1}$$

  These connections ease optimization by allowing gradients to flow around sub-layers.

Overall, the Transformer encoder and decoder stacks alternate these layers. This architecture has no recurrent units, which allows much greater parallelization and the ability to learn long-range dependencies efficiently. The image above illustrates an encoder block (left) and decoder block (right), highlighting the multi-head attention and feed-forward modules.

2.1.2 Encoder–Decoder vs Decoder-Only Models

Transformer-based LLMs come in two main variants:

- **Encoder–Decoder Models (Seq2Seq, e.g. T5, BART):** These have a distinct encoder and decoder. The encoder ingests the input text and the decoder generates the output text autoregressively. Pre-training often involves a denoising or "text-to-text" objective. For instance, BART [3] uses a bidirectional encoder and a left-to-right decoder, effectively generalising both BERT and GPT within one model. Such models excel at conditional generation tasks: translating or summarizing a given input, answering questions by reading contextand other sequence-transduction tasks. In particular, BART achieved state-of-the-art results on abstractive summarization, dialogue generationand question answering, significantly outperforming prior systems (e.g. up to +6 ROUGE on summarization benchmarks). Likewise, T5 [4] casts all problems as text-to-text and obtains SOTA on many NLP tasks by fine-tuning its encoder–decoder network.
- **Decoder-Only Models (Autoregressive, e.g. GPT series):** These use only the transformer decoder stack with masked self-attention and no

separate encoder. They model text generation by predicting the next token given all previous tokens. Architecturally this is exactly the left-to-right branch of a Transformer. Decoder-only LMs are trained on large corpora to predict the next word and thus learn broad language patterns. GPT-3 [1], for example, is a purely decoder-only model with 175B parameters that, without any task-specific fine-tuning, achieves strong results on translation, QA, cloze, arithmeticand even generates realistic news articles. Such models are particularly suited to open-ended generation tasks, chatbots, story or code generation, etc., where text is generated given only a prompt.

In summary, encoder–decoder transformers (T5, BART) are typically used for tasks that require mapping an input sequence to an output sequence, leveraging cross-attention between encoder and decoder. Decoder-only transformers (GPT) are used for unconditional or prompt-driven generation, using only self-attention on past tokens. Each class has shown state-of-the-art performance in its domain.

2.2 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an approach that enhances large language models (LLMs) by integrating an external retrieval step into the generation process. Instead of relying solely on a model's fixed parametric knowledge, a RAG system will retrieve relevant documents (e.g., from a knowledge base or document corpus) based on the user's query and then augment the LLM's input with this retrieved content. This allows the model to produce answers that are more up-to-date, factual and context-specific. In a typical RAG pipeline, a retriever component first finds pertinent text passages for a given query and these passages are fed, along with the query, into the generator which produces the final answer. This retrieve-then-generate paradigm enables the model to access information beyond its internal memory, addressing key limitations of standard LLMs such as hallucinations and outdated knowledge [5]. Figure 2.2 illustrates the classic RAG architecture: a user question is sent to a retriever, which may use a search index or vector store, the top-k relevant passages are returned and the LLM composes an answer conditioned on those passages.

A crucial aspect of RAG systems is the choice of retrieval mechanism. Broadly, retrieval methods are categorized as sparse (lexical) or dense (vector) retrieval, or a hybrid combination of both. Each of these has distinct characteristics, as discussed next.

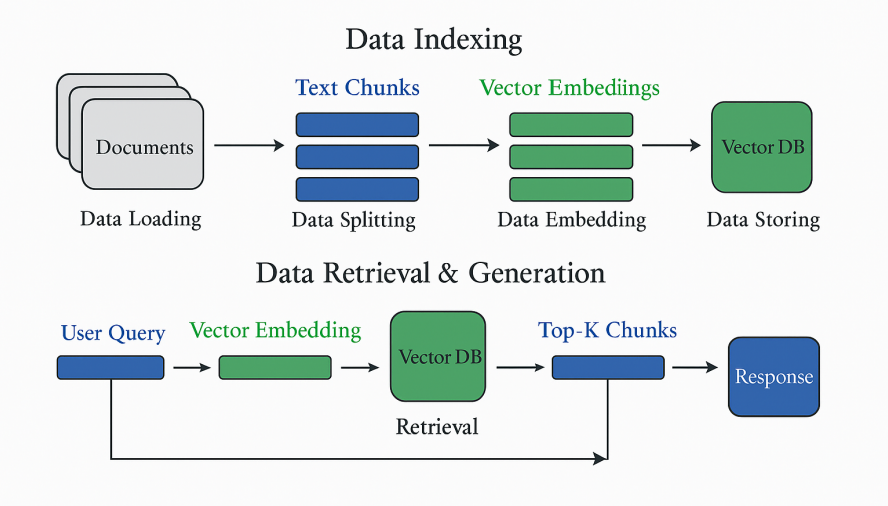

Figure 2.2: Classic Retrieval-Augmented Generation pipeline. A query is used to retrieve relevant text chunks from a knowledge sourceand the language model then generates an answer grounded in the retrieved evidence.

2.2.1 Dense vs. Sparse vs. Hybrid Retrieval

Sparse retrieval methods rely on lexical overlap between the query and documents. Traditional search algorithms like TF-IDF or BM25 represent text as bags-of-words and rank documents based on keyword frequency and exact term matching. Sparse retrieval excels at precise term matching – it is very effective when the query contains specific keywords that should appear in the answer source. However, sparse methods struggle with semantic paraphrases or synonyms. For instance, if a relevant document uses different wording, a purely lexical match may miss it. For example, a BM25-based search might fail to retrieve a passage about "influenza" if the query uses the term "flu." Sparse approaches are fast and interpretable, but their recall is limited by vocabulary mismatch.

Dense retrieval uses vector representations to capture semantic meaning. Techniques such as DPR (Dense Passage Retrieval) map queries and documents into high-dimensional embeddings, often using transformer models, so that semantically related text has closer vectors even if they don't share keywords. A dense retriever can retrieve relevant texts that are paraphrased or conceptually related to the query, overcoming the synonym problem. For instance, a dense model might recognize that a query about "financial well-being" is related to a document about "economic health" despite no literal word overlap. Dense retrieval thus improves recall and semantic relevance. On the downside, it may introduce some irrelevant results if the model's embeddings find loose semantic connections that don't precisely answer the question. Dense methods also typically require neural encoding and

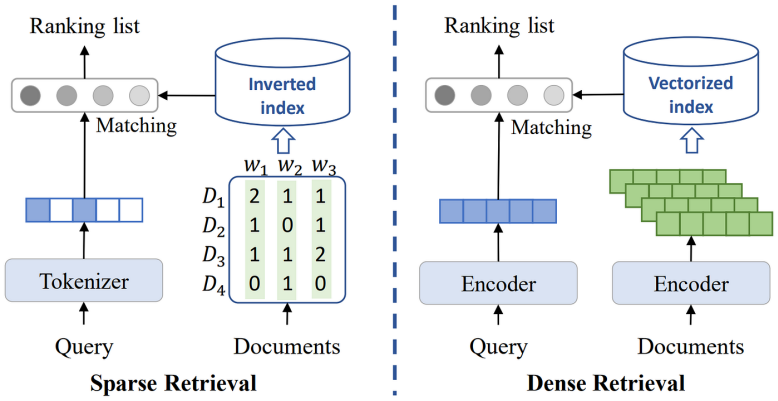

Figure 2.3: Sparse vs. dense retrieval: token-based inverted index vs. embedding-based vector index.

approximate nearest-neighbor search, which can be more computationally intensive than sparse indexing.

Hybrid retrieval aims to get the best of both worlds by combining sparse and dense approaches. In a hybrid RAG system, the query is run through both a sparse retriever (e.g., BM25) and a dense vector retriever in parallel. The results from both are then fused – for example, by taking the union of documents and re-ranking them, or by scoring via a weighted sum of sparse and dense relevance scores. This can significantly improve coverage: the sparse component ensures high precision for on-topic keywords, while the dense component brings in semantically related information that sparse matching might miss. Studies have shown that hybrid retrieval often achieves higher overall accuracy (precision and recall) than either method alone, grounding the generative model in more complete evidence and thus mitigating knowledge gaps or hallucinations [6]. Modern RAG implementations often employ hybrid search pipelines (for example, using a BM25 index together with a vector index like FAISS or HNSW and merging their outputs at query time) and this approach has become popular in open-source frameworks (LangChain, Haystack, etc.) for robust retrieval performance even in challenging settings.

In summary, sparse vs. dense retrieval can be seen as keyword matching vs. semantic matching. Sparse methods (e.g., BM25) offer exact precision but limited semantic reach, while dense methods (e.g., DPR or other embedding-based search) offer semantic generalization but may retrieve tangential content. Hybrid retrieval marries the two, using keywords for precision and vectors for meaning, thereby delivering more relevant and diverse supporting passages. Many classic RAG systems can be characterized by which retriever they use: for example, the original RAG model by Lewis et al. (2020) used a dense retriever (DPR) on Wikipedia [7], while other QA systems have used sparse retrievers or hybrids. Choosing the right

retrieval strategy (dense, sparse, or hybrid) is crucial for effective RAGand often depends on the nature of the data and queries.

2.3 Iterative vs. Parallel Retrieval-Generation Interactions

Beyond the one-step "retrieve then generate" process of classic RAG, researchers have explored more interactive architectures for how the retriever and language model work together. A recent survey defines Retrieval-Augmented Language Models (RALMs) in terms of how many interactions occur and in what manner [8]:

2.3.1 Single-turn (one-shot) interaction.

This is the standard RAG approach described above – the retriever provides documents once, then the LLM produces an answer using that static retrieved context [7]. There is no feedback loop; the process is sequential and concludes after the generation. This works well for straightforward questions answerable with a single set of documents, but it may falter if the query is complex and requires multiple hops or clarifications.

2.3.2 Sequential multiple interactions.

In this iterative setup, the retriever and generator engage in a multi-turn dialogue. The LLM may pose follow-up queries or reformulate the original query based on partial answers and then retrieve again, multiple times, before final answer generation. Essentially, the system can refine its context step by step, handling complex or ambiguous queries through iterative retrieval. For example, the LLM might initially retrieve some information, then realize a detail is missing and issue a secondary query to the retriever, possibly conditioned on the first result, to gather more details and so on. This approach allows multi-hop reasoning where the answer is assembled from information across multiple sources or reasoning steps. Sequential iterative retrieval has been used in QA systems that require reasoning over multiple documents or in agent-like systems that plan a chain of retrieval actions. The advantage is a more thorough exploration of the knowledge base, often yielding more complete answers for complex tasks. The downside is increased latency and complexity – multiple LLM calls and retrieval rounds can be slow and harder to orchestrateand there is a risk of error propagation if an early retrieval is off-track.

2.3.3 Parallel interaction.

Here, the retriever and the language model work independently on the user's query and their outputs are later combined in some fashion. In other words, the LLM might generate an answer from its internal knowledge while in parallel a retrieval component fetches relevant text and finally the two results are fused to produce the final output. One notable example of this paradigm is the kNN-LM by Khandelwal et al., which augments a pretrained language model with a k-nearest-neighbor search over an external datastore [9]. In kNN-LM, a base LM generates a probability distribution for the next word while simultaneously retrieving the nearest neighbor contexts from a large text index; the two distributions (LM and nearest-neighbor) are then interpolated at each step to guide generation. Khandelwal and colleagues showed that this method significantly improved perplexity and helped the model produce factual or rare outputs by copying from retrieved examples. Parallel RALM designs are less common than sequential ones, but they can be advantageous in certain scenarios – for instance, to leverage the fluency of a powerful LLM while also grounding it with facts from a retriever, without the retriever directly dictating intermediate steps. The parallel approach can reduce sequential dependency, potentially improving efficiency since retrieval does not have to wait on generation or vice-versa. However, designing a good merging or interpolation mechanism is non-trivial and there is a risk that the final answer may mix retrieved facts and learned content incoherently if not carefully coordinated.

Overall, these RALM interaction modes offer different trade-offs. Sequential iterative retrieval is powerful for complex queries that need stepwise reasoning or disambiguation, where the LLM can progressively focus intently on the answer, whereas parallel retrieval-generation can leverage complementary strengths of two systems but requires careful fusion. Many advanced systems, such as agent-based RAG frameworks, utilize sequential multiple interactions – effectively treating the LLM as an "agent" that can plan a series of retrieval and reasoning actions. Recently, the concept of *agentic RAG* has emerged for highly complex tasks: it embeds an autonomous agent into the RAG pipeline to plan and manage these sequential retrieval steps, offering even more flexibility. Agentic RAG systems can dynamically decide how many retrievals to perform or which tool to use at each step, but they inherit all the challenges mentioned above (coordination overhead, potential error cascades, etc.). We now turn our attention to a different direction of RAG evolution: integrating structured knowledge in the form of graphs.

2.4 Pros and Cons of Classic vs. Iterative RAG Approaches

To better understand the evolution of RAG systems, we summarize the advantages and disadvantages of the classic one-shot RAG (single retrieval + generation) versus iterative RAG (multi-step or agentic retrieval-generation cycles):

Classic RAG (Single-Step Retrieval)

Advantages: Simplicity and speed – a one-step RAG pipeline is straightforward to implement and typically faster at inference time since it only retrieves once and generates once. This approach is often sufficient for factoid questions or isolated queries where a single document contains the answer. It minimizes complexity and potential error propagation. Classic RAG also keeps the answer grounded in a small set of documents, which can help with answer focus.

Disadvantages: Limited ability to handle complex queries that require synthesizing information from multiple sources or doing reasoning over several pieces of evidence. If the first retrieval misses some relevant information or retrieves irrelevant context, the system has no opportunity to recover or refine its results. Classic RAG systems can struggle with context integration, sometimes producing fragmented or overly generic answers because the single retrieved batch may not fully cover the query's nuances. They are essentially static, lacking feedback loops to correct or improve upon an initial attempt. This makes them less effective for multi-hop questions, such as "Find X and then based on that find Y", or for disambiguating user queries that are broad or vague. In summary, one-shot RAG is brittle when faced with complex information needs beyond simple lookup.

Iterative/Agentic RAG (Multi-Step Retrieval)

Advantages: Adaptability and thoroughness. Iterative RAG systems can tackle complex or exploratory queries by refining their search over multiple turns. The LLM can analyze retrieved results and issue follow-up queries, enabling multi-hop reasoning and deeper contextual understanding. This often leads to more comprehensive answers, as the system can gather diverse pieces of evidence and combine them. Such systems shine in scenarios like open-domain QA, complex problem solving, or interactive dialogue, where responses benefit from dynamic planning. Moreover, an agentic approach can incorporate tools and operations by calling external APIs or performing calculations, along with retrieval, greatly extending the system's capabilities. In essence, iterative RAG introduces a feedback loop that can improve relevance and correctness with each step [8].

Disadvantages: Increased complexity in design and runtime. Coordinating multiple retrieval and generation steps, often with an autonomous agent or controller, requires sophisticated logic for decision-making, stopping criteria and integrating partial results. This complex orchestration can be hard to get right and may introduce new failure modes. For example, the agent might get stuck in a loop or pursue a wrong line of inquiry. Additionally, multi-step interactions incur higher latency and computational costs, because the model is invoked multiple times and numerous documents may be retrieved across turns. There are also scalability concerns, as each step adds overhead, serving many queries in real time becomes challenging. Finally, while iterative systems aim to mitigate errors by refinement, if not carefully managed they might compound errors (a wrong assumption in step 1 could lead the agent astray in step 2, etc.). Thus, although iterative RAG can yield superior results on hard tasks, it demands careful engineering and more resources.

In practice, a spectrum of solutions exists between these extremes. Practitioners often start with a classic RAG baseline and only add iterative retrieval if needed due to query complexity. Agentic RAG approaches, which dynamically plan retrieval and reasoning steps, are at the cutting edge but come with the aforementioned coordination overhead. Next, we explore a different direction in RAG: leveraging structured knowledge via graphs.

2.5 GraphRAG and Knowledge Graph-Based Retrieval

A notable extension of the RAG paradigm is the incorporation of knowledge graphs as a way to represent and retrieve information. *GraphRAG* refers to RAG systems that leverage a graph-structured knowledge base (nodes and edges representing entities and their relations) alongside or instead of free-text documents. The key idea is that a graph can encode relationships and global structure in the data, enabling the system to perform relational reasoning and "connect the dots" between pieces of information more effectively than retrieving isolated text chunks. This is especially useful for queries that involve complex relationships or require a global understanding of a domain (e.g., questions about how multiple entities are interrelated).

In a GraphRAG system, the knowledge source is not just an unstructured text index or embedding store, but a *knowledge graph (KG)* constructed from the corpus. Nodes in this graph typically represent important entities and edges represent relations between entities (e.g., Person A *works at* Company B, or Event X *occurred in* Year Y). The RAG pipeline is modified to utilize the graph: for example, the retriever might perform a graph traversal or a structured query on the KG to find relevant subgraphs or linked entities related to the question. The results of the

graph query can then be converted into textual form (or some structured form) for the LLM to consume, or the LLM might even interact with the graph directly through a graph-aware QA chain. In essence, GraphRAG provides an alternative retrieval mechanism where connectivity and relationships between facts can be exploited.

2.6 Building the Knowledge Graph.

Constructing a knowledge graph from unstructured text is a non-trivial step that typically involves NLP techniques for information extraction. A simple approach is to use Named Entity Recognition (NER) to detect entity mentions in textand then connect entities that co-occur in the same context, assuming they are related. For instance, one can create a co-occurrence graph where any two entities mentioned in the same sentence or subsection are linked by an undifferentiated relationship (e.g., a "co-occurs-with" edge). This yields an undirected graph indicating associations, though without explicit relation types. Although they are easy to create, co-occurrence graphs do not provide explicit semantics. This means that while they can reveal clusters of related concepts, they do not show which entities are connected, or why. A more sophisticated pipeline will attempt to identify specific relations between entities. This can be done via rule-based or ML-based relationship extraction, for example recognizing patterns like "X was born in Y" to create a triple (X — `born_in` → Y) instead of a generic link. Modern approaches often employ transformers or even LLMs fine-tuned for information extraction to produce structured triples from text, as seen in Open Information Extraction systems or with tools like LangChain's `LLMGraphTransformer`. The end result of these processes is a knowledge graph that might contain various node types (Person, Organization, Location, etc.) and relation types (`located_in`, `works_for`, `part_of`, etc.), providing a rich semantic network of the source data. In practice, building a high-quality KG requires careful design, since it may involve entity linking to canonical databases, coreference resolution to merge mentions referring to the same real-world entity and ensuring important relations are captured. In the context of GraphRAG retrieval, the completeness of the graph is crucial: if an entity or connection is missing from the KG, the system might overlook relevant information.

2.7 Graph-Based Retrieval.

Once a knowledge graph is in place, a GraphRAG system can use it in several ways. A straightforward method is *graph traversal*, which given a query, identify one or more "focus" entities mentioned in the query, then traverse the graph to find

connected entities or a subgraph that could contain the answer. For example, if the question is "What are the main themes connecting Alice's and Bob's research?", a traversal might find that Alice and Bob are linked via the node *Quantum Computing*, indicating both have work related to that theme. The system could then retrieve textual information about that connecting node, in this example summaries or documents about *Quantum Computing*) to help answer the question. Another approach is to treat the knowledge graph as a database and use structured queries (e.g., SPARQL or Cypher) to directly retrieve facts or paths relevant to the query. There are also specialized QA chains that integrate graphs, such as feeding the LLM a representation of the subgraph or having the LLM issue graph queries via a tool. The advantage of graph-based retrieval is that it exploits relational structure: the system can reason not just about isolated facts but about how facts interrelate. This is crucial for queries that involve multiple entities or require understanding the interplay between concepts. For instance, "How is X connected to Y?" or "Summarize the relationship among A, Band C". GraphRAG inherently enables multi-hop reasoning, since following a path in the graph is like traversing a chain of reasoning steps connecting different pieces of information.

However, GraphRAG also brings challenges. Building and maintaining the graph adds an extra layer of complexity and processing overhead. There is an inherent coverage issue, when a fact or entity is not represented in the graph, the system might miss it even if it exists in the raw text. This is why GraphRAG is often combined with text vector retrieval as a fallback. Performance-wise, querying a large graph can become slow if the graph has millions of nodes/edges and memory usage can be significant. The approach is only as good as the quality of the graph: incorrect or spurious relations in the KG can mislead the modeland conversely, missing links can lead to incomplete answers. Integrating unstructured text with the graph is non-trivial, and a hybrid GraphRAG strategy is often the best practice, using the knowledge graph for structured relations and a vector index for general text, with the results merged successively. Despite these challenges, GraphRAG has shown to be highly effective in certain domains, such as biomedical research or legal corpora, where there are many entities and relationships and where queries require a more global or relational understanding of the data.

2.8 Pros and Cons of GraphRAG

To summarize the utility of graph-based RAG systems, we can list their main advantages and disadvantages:

Advantages: GraphRAG enables richer reasoning over relationships. By leveraging node connectivity and hierarchies, it provides context that spans multiple documents or facts, which a flat text retrieval might miss. It is particularly powerful

for questions requiring relational understanding (e.g., "How are these entities related?") or sense-making across a large corpus, because the graph structure helps the model organize information and avoid getting lost in irrelevant details. Additionally, GraphRAG can help reduce hallucinations and improve factuality, since the existence of an edge between nodes is a constrained, verifiable piece of data and the model can be guided to stick to known relations in the graph. The graph can also act as a form of compressed memory or summary of the corpus, which is useful for very broad or exploratory queries. In this way, rather than retrieve dozens of raw documents for a broad question, the system might retrieve a concise subgraph or a pre-computed summary of a cluster of entities, making it easier for the LLM to handle. In essence, GraphRAG offers a way to inject structured knowledge into the generation process, which is beneficial for domains where structure matters. For example, in an enterprise setting, a knowledge graph might encode organizational information or product ontologies that the LLM can leverage when answering questions.

Disadvantages: The need to construct and maintain a knowledge graph means higher development and maintenance effort. Automatic graph construction can be error-prone. for instance, extracting incorrect relations or merging unrelated entities will introduce noise that can confuse the generator. There is also an inherent trade-off in graph design: a naive co-occurrence graph might include too many loosely related connections yielding noise, whereas a strictly curated graph might be too sparse and miss relevant links. Moreover, not all queries benefit from a graph; for simple fact lookups, a graph can be overkill and a direct text retrieval might suffice. Performance and scalability are concerns: large graphs require efficient query algorithms and indexing, and they may not scale well to web-sized knowledge without significant engineering, though techniques like graph databases and community detection can help manage this. Thus, GraphRAG tends to excel in specialized scenarios but may not universally outperform standard RAG on every task. It's a tool best used when the domain naturally contains a web of relationships that the QA process can exploit, or when answers require synthesizing information across many related facts.

2.9 Outlook: Microsoft's GraphRAG System

Graph-based RAG has garnered enough interest that major research efforts have been dedicated to it. Notably, in 2025 Microsoft introduced a GraphRAG approach for query-focused summarization over large text corpora [10, 5]. In their method, an LLM is used to build a knowledge graph from a collection of documents: the LLM identifies key entities and relationships to form the graph, then automatically generates hierarchical community summaries for clusters of related entities in a

bottom-up fashion, using a community-detection algorithm such as Leiden to group the clusters [11]. At query time, those community summaries, which capture high-level themes in the data, are leveraged in a map-reduce style QA process: partial answers are generated from each relevant community and then merged into a final answer. This GraphRAG system is designed to handle broad, analytical questions like "What are the main themes in this dataset?", which conventional vector-based RAG struggled with. The results reported by Microsoft's team showed substantial improvements over a standard vector-only RAG baseline in terms of answer comprehensiveness and diversity. In other words, by using a knowledge graph and hierarchical summaries, the GraphRAG approach provided more complete and insightful answers to complex queries than a classic retrieve-and-read pipeline [5]. In the next chapter, we will delve deeper into the design and evaluation of Microsoft's GraphRAG system, examining how it implements the concepts outlined here and how it compares to traditional RAG techniques.

2.10 Evaluation of RAG Systems: Benchmarks and the RAGAs Framework

Evaluating the performance of retrieval-augmented generation systems is inherently challenging, since it requires assessing not only the quality of the retrieved context but also the correctness and groundedness of the generated answer. Traditional NLP metrics such as BLEU, ROUGE, or simple retrieval recall only capture one side of this process: either how well the model reproduces reference answers, or how many relevant documents were retrieved. However, none of these approaches fully reflect whether a RAG system truly understands and integrates retrieved evidence into its output [12]. Recent research has thus emphasized the need for holistic evaluation frameworks and benchmarks specifically designed for RAG models.

Among these, **RAGAs** (Retrieval-Augmented Generation Assessment) has emerged as one of the most widely adopted and flexible solutions [13]. Introduced in 2024, RAGAs provides an automated, LLM-based evaluation framework capable of judging a RAG system's performance without requiring human-written reference answers. Instead, it employs a set of complementary metrics that jointly assess both retrieval and generation quality. The key evaluation dimensions include: *faithfulness*, which measures whether the model's answer remains factually grounded in the retrieved context; *answer relevancy*, which gauges how well the answer addresses the user's query; and *context precision and recall*, which evaluate the relevance and completeness of the evidence used.

One of the main advantages of RAGAs is its reference-free design, which allows it to be applied even to domains where no gold-standard answers exist, such as enterprise documentation or proprietary knowledge bases. By leveraging large

language models as evaluators, RAGAs approximates human judgment in assessing factuality, relevance and grounding, providing a practical way to benchmark RAG systems quickly and consistently.

In this work, we employ RAGAs to compare our graph-augmented RAG framework against a baseline on a standard open-domain dataset. This experiment serves as an initial validation step, allowing us to quantify the benefits of introducing structured retrieval before deploying the system in an enterprise documentation environment. RAGAs' holistic perspective ensures that the observed improvements in accuracy also correspond to more grounded and contextually supported answers, aligning the evaluation with the ultimate goal of reliable knowledge retrieval. , Finally, we introduce the graph-based GraphRAG variant (§3.4), describing its knowledge-graph index, community hierarchy and the implementation extensions developed in this thesis.

Chapter 3

System Implementation

3.1 Introduction to the Implemented RAG Variants

Retrieval-Augmented Generation (RAG) is an approach that integrates information retrieval into the workflow of a language model to augment its knowledge and reduce hallucinations. In this chapter, we present the implementation details of three baseline RAG systems: a **dense retrieval RAG**, a **sparse retrieval RAG** and a **hybrid retrieval RAG**.

These systems serve as baselines for comparison and ablation, each employing a different strategy for fetching relevant context to supply to the Large Language Model (LLM) before generation. The dense and sparse variants represent two paradigms in information retrieval, which are neural semantic search vs. classical lexical search, while the hybrid variant combines both to leverage their complementary strengths. Our implementation is built using the LangChain framework, which provides modular components for document loading, chunking, embedding and chaining together the retrieval and generation steps. We describe each of these components and how they form a complete RAG pipeline. The more advanced **GraphRAG** system, which incorporates graph-based retrieval, will be detailed after, allowing us to focus first on the baseline methods without graph augmentation. By examining these baseline RAG variants, we set the stage for understanding the improvements introduced by the GraphRAG approach.

Each RAG variant in this chapter follows the general RAG architecture introduced by Lewis et al. [14]: a question or user query is expanded with additional context retrieved from an external knowledge source and this augmented query is then passed to an LLM to produce a final answer. The key difference between the variants lies in *how the relevant context is retrieved*. Below, we outline the libraries and components used in our implementation (§3.2), the embedding model

and vector store for dense similarity search versus the $BM25$ algorithm for sparse search (§3.2.2), the LLM configuration (§3.2.3) and the prompt design that ensures the LLM uses the retrieved evidence (§3.2.4). We then detail the end-to-end implementation of each of the three baseline RAG systems in turn (§3.3.1–§3.3.3), including code-level insights into how documents are chunked, how retrieval is performed and how the retrieved results are integrated into the generation pipeline.

3.2 Libraries, Chunking and Embeddings

Before diving into each RAG variant, we describe the common tools and techniques used across our implementations. This includes the LangChain libraries [15] and components for document handling and retrieval, the embedding model and vector database for dense search and how we perform text chunking to manage LLM context lengths.

3.2.1 LangChain Components

Our implementation makes extensive use of the LangChain framework, including the `langchain` and `langchain_community` modules, to simplify the construction of the RAG pipelines. Key components utilized are:

- **DirectoryLoader** and **TextLoader**: These loaders facilitate ingesting a corpus of documents from local files. `DirectoryLoader` can traverse a directory and apply `TextLoader` to each file, reading its contents into a standardized `Document` object. This allows us to easily load all knowledge source files into memory for further processing.
- **CharacterTextSplitter**: After loading, each document is split into smaller chunks using `CharacterTextSplitter`. We configure this with a suitable chunk size, which in our case was 1200, and optional overlap, which we set to 100. Chunking ensures that each piece of text fits within the context window of the LLM and is focused on a specific subtopic. By splitting documents into chunks, we improve retrieval granularity, allowing the retriever to find and return only the most relevant portions of text for a query.
- **OpenAIEmbeddings**: This module provides an interface to OpenAI's text embedding model for generating dense vector representations of text. We use `OpenAIEmbeddings` to encode each text chunk and queries into a high-dimensional embedding space suitable for semantic similarity search.
- **Chroma**: Chroma is an open-source vector store that we use to index and query embeddings. After computing embeddings for all document chunks, we

store them in a Chroma database. Chroma provides fast similarity search over these vectors, returning document chunks that are nearest to a given query vector.

- **BM25Retriever**: This component implements a classical sparse retrieval algorithm $BM25$ [16] for our corpus. This sparse retriever indexes the text chunks by their terms and allows querying with a new text query to obtain relevant documents scored by BM25 (a TF–IDF-based ranking function). We use the `BM25Retriever` from `langchain_community` to perform lexical matching retrieval without embeddings.

- **EnsembleRetriever**: The ensemble retriever enables combining multiple retriever results into one. In our hybrid RAG system, we initialize an `EnsembleRetriever` with two constituent retrievers, usually a dense one and a sparse one. This component will call each retriever for a query and then merge and re-rank their results into a single list. LangChain's ensemble uses a weighted *Reciprocal Rank Fusion (RRF)* strategy [17] by default to aggregate results, which tends to improve recall by including documents that either method found highly relevant.

- **ChatPromptTemplate**: This is a templating utility to build the prompt, consisting of system/user messages, for the chat-based LLM. We define a prompt template that includes placeholders for the retrieved context and the user's question. So, the `ChatPromptTemplate` fills in these slots at runtime to produce the final prompt message sequence for the LLM. The usage of a prompt template helps enforce a consistent format for queries, ensuring the model is instructed properly to use the context.

- **RunnablePassthrough**: This is a utility that simply passes its input through unchanged. We use `RunnablePassthrough` in constructing the retrieval-generation pipeline to feed the original question into multiple components in parallel. For example, in the hybrid chain, we pass the user question both to the retriever to fetch documents and directly into the prompt template to fill the question placeholder, ensuring the question text is preserved for the LLM input while retrieval occurs.

- **StrOutputParser**: After the LLM produces an output, which in LangChain may be a message object, the `StrOutputParser` is used to extract the raw text string of the answer. This is essentially a post-processing step to ensure the final output of the chain is a clean text answer without additional metadata.

Together, these components allow us to construct a flexible pipeline: we load and chunk documents, embed or index them, perform retrieval, assemble a prompt

with retrieved context, call the LLM and finally extract the answer. LangChain's abstractions simplify each step, while still giving fine-grained control over the process.

3.2.2 Embeddings, Vector Store and Similarity Search

For the dense retrieval variant, we rely on a neural embedding model and a vector database, whereas the sparse variant uses the BM25 algorithm. Here we outline the differences and how the hybrid combines them.

OpenAI Embeddings (`text-embedding-3-small`): We use

`text-embedding-3-small`

an OpenAI's model to convert text into a dense vector representation. This model produces a fixed-dimensional embedding for any given text input, capturing semantic meaning such that similar meanings correspond to nearby points in the vector space. Using dense embeddings for retrieval allows the system to find relevant documents even when they do not share obvious keyword overlap with the query. For example, the query "What is the revenue of the company?" could still retrieve a passage stating "The company reported $X in annual sales," because the embeddings of "revenue" and "sales" would be close in the vector space. This semantic search capability is a key advantage of dense retrieval methods and has been shown to significantly outperform traditional keyword search on many knowledge-intensive QA tasks [18]. In our implementation, after chunking the documents, we pass each chunk through the OpenAI embedding model, via `OpenAIEmbeddings`, to obtain its vector. All vectors are stored along with metadata (e.g. the text content or source of the chunk) in the **Chroma** vector store. The Chroma store enables efficient similarity search: given a new query, we embed the query using the same model and ask Chroma for the top-k most similar chunk vectors. The similarity metric is **cosine similarity** on the embeddings. The result is a list of retrieved chunks ranked by semantic relevance to the query.

BM25 Sparse Retrieval: The second approach uses sparse vector space modeling of text based on token occurrence. We employ the BM25 algorithm [16], a well-known method from information retrieval that scores documents for a query based on term frequency, which is how often query terms appear in the document and inverse document frequency, increasing the weight of rarer terms. Unlike the dense approach, BM25 does not require any model training or embeddings; it operates directly on the text. We initialize `BM25Retriever` with our collection of document chunks. For example, in code we call something like:

```
sparse_retriever = BM25Retriever.from_documents(chunks).
```

This call likely builds an internal inverted index of terms to chunk IDs and computes the necessary IDF statistics for BM25. The retriever uses the library `rank_bm25` under the hood to support BM25 scoring. There is no external store or database needed, ibut only an in-memory index since our corpus is of a manageable size. If the corpus were huge, one might use Elasticsearch or Whoosh, but LangChain's BM25Retriever is a simple solution for moderate data sizes.

At query time, the BM25 retriever tokenizes the user's query and computes a BM25 score for each chunk's text, returning the top k chunks with highest scores. Sparse retrieval excels at finding documents that contain the exact keywords or phrases used in the query. For example, if the question asks for a specific name or code, BM25 is likely to surface documents containing those exact tokens, which a dense method might miss if those exact terms are rare or out-of-vocabulary for the embedding model. However, sparse methods struggle when the query and document use different wording, such as synonyms or paraphrases, a scenario where dense embeddings shine. In fact, dense and sparse retrievers often find *complementary* information [19]: each can retrieve some relevant items the other might overlook.

Hybrid Retrieval (Dense + Sparse Ensemble): To capitalize on the complementary nature of dense and sparse search, our third variant uses a hybrid approach. We combine the above two methods using the `EnsembleRetriever` which invokes both a dense retriever and the BM25 retriever and then fuses their results. In practice, we configure two retrievers: one backed by the Chroma vector store for dense similarity search and one BM25. The ensemble retriever calls both with the user query and obtains two ranked lists of document chunks. These lists are then merged using a rank fusion strategy. LangChain's default ensemble strategy is an implementation of *Reciprocal Rank Fusion (RRF)* [17], which assigns each document a score based on the reciprocal of its rank from each method and then produces a combined ranking. RRF is a simple yet effective technique that has been shown to outperform more complex learned rank aggregation methods in many cases [17]. Intuitively, this means that if a document is highly ranked by either the dense or sparse retriever, it will appear near the top of the merged list. The hybrid approach often yields better coverage of relevant information than either method alone: semantic search finds conceptually related content, while keyword search ensures exact matches are not missed. Recent research on hybrid retrieval confirms that integrating classical and neural IR techniques can significantly improve overall retrieval performance [19]. In our system, we give equal weight to dense and sparse results, though weights can be tuned if desired and we typically retrieve a few results from each before fusion. The end result is a set of top-k context chunks that reflect both semantic relevance and lexical matching to the query.

By switching between these retrieval strategies, we can observe their different behaviors. All three variants feed their retrieved documents into the same generation module, composed of LLM + prompt, which ensures a fair comparison of the

retrieval component's effect on the final answer quality.

3.2.3 LLM Configuration

For the answer generation step, we use an OpenAI Chat model via LangChain's `ChatOpenAI` interface. The model configuration is kept consistent across all RAG variants for fair comparison. The specific settings used are:

- **model_name = "gpt-4o-mini"** – This refers to the language model we query for generation. In our implementation, we used a GPT-4 class model, denoted here as "`GPT-4o-mini`", which is a smaller variant of OpenAI's GPT-4. This model was chosen to balance performance with computational constraints. It retains the advanced reasoning and understanding capabilities of GPT-4, but with a reduced size to allow faster responses in our development environment. The key is that all variants use the same model, so any differences in answer quality can be attributed to differences in retrieval, not the LLM itself.
- **temperature = 0** – We set the generation randomness to zero. A temperature of 0 forces the model to pick the most probable next token at each step, essentially making the output deterministic given a fixed prompt. This is ideal for question-answering tasks where we want factual, reproducible answers and minimal creativity. A temperature of 0 significantly reduces variability and the chance of hallucinating details, as the model will stick closely to the highest-confidence answer (often drawn from the provided context).
- **max_tokens = 512** – This limits the length of the model's answer to at most 512 tokens. This value is chosen to ensure the answer can be sufficiently detailed if needed, but not excessively long. Moreover, it also guarantees the response fits within the context window alongside the prompt. By capping the output length, we prevent the model from rambling or going off-topic and avoid hitting token limits that could truncate the answer mid-sentence.
- **timeout = 60** – We specify a timeout of 60 seconds for the model API call. This is a safety measure so that if the model or network is unresponsive, the system doesn't hang indefinitely. In practice, the model typically responds much faster, but the timeout ensures the pipeline fails gracefully and can retry or exit if something goes wrong.
- **retry_settings = `{"max_retries": 3}`** – LangChain's retry mechanism is enabled to automatically handle transient errors. If the LLM API call fails due to a rate limit, network glitch, or other error, the chain will retry up to 3 times. This improves robustness of our system, as occasional API failures will not result in missing answers but will be retried after a short backoff. With

`max_retries=3`, we give the model multiple chances to return a result, which is usually enough for any temporary issue to resolve.

In summary, our LLM is configured for reliable, factual generation: using a strong GPT-4-based model, disabling randomness to get consistent outputs grounded in the input context, limiting answer length and adding safeguards for timeouts and retries. These settings ensure that when provided with relevant retrieved documents, the model focuses on generating a correct answer from them rather than producing creative or unsupported content. All three RAG variants use this identical LLM setup, so the differences in their performance can be traced back to the retrieval stage rather than any variation in generation parameters.

3.2.4 Prompt Design

The three baseline systems share the same prompt template, which is designed to make the use of retrieved context explicit, constrain the model to this context and provide a simple, uniform interface across all retrieval variants. The template used in our implementation is the following:

```
You are an assistant for question-answering tasks.
Use the following pieces of retrieved context to answer the question.
[The exact answer is less than 5 words.] (for the hotpotQA dataset)
If you don't know the answer, just say that you don't know.
Do not add information from outside the documents.
Question: {question}
Context: {context}
Answer:
```

In the LangChain implementation, this template is represented as a chat-style prompt with two logical parts:

- **System message.** The initial instructions (up to "Do not add information from outside the documents.") are encoded as a system message. They define the assistant's behaviour: rely on the retrieved context, avoid external knowledge and explicitly answer "I don't know" when the context does not support a grounded answer. This framing is intended to reduce hallucinations and aligns with established practice in retrieval-augmented generation [14].
- **User message.** The remaining fields, which are `Question: {question}`, `Context: {context}` and `Answer:` are encoded as the user message. At runtime, `{question}` is filled with the user query, while `{context}` is populated by concatenating the top-k retrieved chunks returned by the chosen retriever. The trailing `Answer:` token signals to the model where to begin its response.

This prompt design ensures that the model has all relevant information explicitly exposed when formulating an answer. By presenting the question and the retrieved context together within a constrained instruction, the model is reminded to ground its output in the provided evidence rather than performing open-ended generation. The requirement not to add information from outside the documents, combined with the permission to answer "I don't know", encourages conservative behaviour when the retrieval step fails or is incomplete. Empirically, this configuration proved stable across all three RAG variants and allows a fair comparison focused on differences in retrieval rather than prompt engineering.

3.3 Implementation of the Three Baseline RAG Systems

We now describe how each of the three baseline RAG systems is implemented in our project. Each system follows the general pipeline of document preprocessing, retrieval, prompt construction and LLM query, but they differ in the retrieval component. We highlight the code-level details of how documents are prepared, how each retrieval method is set up and invoked and how the results are fed into the LLM. Throughout these implementations, the LLM model and prompt template remain the same as described above, isolating the effect of the retrieval strategy on the end result.

3.3.1 Dense Retrieval RAG

The dense retrieval RAG system uses semantic vector similarity to find relevant context. Its implementation pipeline can be summarized in the following steps:

1. **Document Loading:** We use `DirectoryLoader` to load all documents from a specified folder, such as a knowledge base of text files. The directory loader internally uses `TextLoader` for each file, resulting in a list of `Document` objects. Each `Document` contains the raw text and associated metadata, like file name or other identifiers which we can use later if needed for source attribution.

2. **Text Chunking:** The loaded documents, which might be several paragraphs or pages each, are split into smaller chunks using `CharacterTextSplitter`. In our notebooks, we configured the splitter with a chunk size that balances informativeness with brevity, splitting into approximately 1000 characters per chunk with some overlap of 100 characters to avoid cutting important context between chunks. This produced a list of chunked `Document` objects. Each chunk has its own text, which is a substring of the original document and retains metadata linking it back to the source. Chunking is crucial for dense

retrieval because it increases the likelihood that any given query will have a closely matching chunk, whereas matching to a long document only partially would ruin the utility of the RAG system. It ensures the LLM is not given an entire long document when only a part is relevant.

3. **Embedding and Indexing:** Next, we generate embeddings for each chunk. We initialize an `OpenAIEmbeddings` instance, with the API credentials and model `text-embedding-3-small`, and call it on each chunk's text. This yields a high-dimensional vector for every chunk. We then create a `Chroma` vector store, adding all chunk vectors to it. In code, this can be done by something like:

```
embedding = OpenAIEmbeddings(model="text-embedding-3-small"))

vector_store = Chroma.from_documents(chunks, embedding)
```

   This builds an index where each entry is the embedding of a chunk and the content-metadata of that chunk. At this stage, we effectively have our knowledge corpus in an indexed form suitable for fast nearest-neighbor search in the embedding space.

4. **Retriever Setup:** We obtain a retriever interface from the vector store by calling `vector_store.as_retriever()`. We configure this retriever with a number of results to return, say $k = 4$ or 5, meaning it will fetch the top 4-5 most similar chunks for any query. The retriever uses cosine similarity under the hood and can optionally use maximal marginal relevance MMR to diversify results, though in our base implementation we primarily used straight similarity ranking.

5. **Query Retrieval:** When the user poses a query, we feed that query string into the retriever: `retriever.get_relevant_documents(query)`. The retriever computes the embedding for the query, using the same OpenAIEmbeddings model, and compares it to all stored chunk embeddings in Chroma. It then returns the top k `Document` chunks that have the highest similarity scores. These returned Document objects contain the chunk text. It is also possible to use the metadata if needed, but primarily the text content is needed as context.

6. **Prompt Construction with Context:** The retrieved chunk texts are then compiled into the prompt template. Using `ChatPromptTemplate`, we fill in the {context} placeholder with the text of these k chunks. In our implementation, we joined the chunks either as bullet points or separate paragraphs in the context section of the prompt. We also fill in the {question} placeholder with the user's question. The result is a fully formatted prompt, altogheter with

system instructions and user message containing context and question, ready for the LLM. For example, if the question was "What causes rainbows?" and the retrieved chunks discussed light refraction and prism experiments, those chunk texts would appear under "Context:" and then the question is asked.

7. **LLM Generation:** We call the `ChatOpenAI` model with the constructed prompt. In LangChain, this could be done by creating a `LLMChain` or using the prompt's `format()` and then passing to `llm({})` directly since our prompt is a single-turn. The model, with temperature 0, processes the context and question and generates an answer. Because of the prompt design, the answer should ideally be drawn from the given context chunks.

8. **Output Parsing:** Finally, we apply `StrOutputParser` to the model's response to extract the answer text. The output is then a plain string, which we can display to the user or evaluate.

Throughout the above process, the role of LangChain's `RunnablePassthrough` is worth noting: in our actual notebook, we constructed a chain using LangChain's "Runnable" pipeline paradigm. Essentially, we created a `RunnableSequence` that tied together the retriever and the LLM with the prompt. The `RunnablePassthrough` was used to feed the original query into two parallel branches of this sequence: one branch goes into the retriever to produce documents and another branch passes the query unchanged into the prompt as the {question} input. The final chain looked conceptually like:

```
Question (input) --[Retriever]--> Documents --+
Question (input) --[Passthrough]-> Question --+
+--> [PromptTemplate] -> [LLM] -> [OutputParser]
```

Here, the plus sign indicates that the prompt template receives two pieces of data: the question and the documents and then the LLM generates using both. While this might be an implementation detail, it shows how LangChain allows parallel flows and merging for building a complex chain. In simpler terms, one can imagine the code doing: `retrieve docs = retriever(query)`, then `prompt = prompt_template.format(question=query, context=docs)` and ultimately `answer = llm(prompt)`.

The dense RAG system, implemented as above, ensures that the model is always given semantically relevant context. It takes advantage of the power of neural embeddings to find context that might not share keywords with the question but is topically related. This is particularly useful for questions where the answer might be phrased differently in the source text than the user's question phrasing. By using a strong embedding model, the retriever can handle vocabulary mismatch

issues better than a keyword search. One limitation to note is that if the embedding model is not familiar with very domain-specific terminology or if the corpus is small and the relevant info is very specifically worded, the dense retriever might sometimes return somewhat relevant but not exact passages.

Overall, the final result is an end-to-end dense RAG system: a user asks a question, we retrieve related chunks via vector similarity and GPT-4 answers using those chunks as evidence. This system embodies the approach of Lewis et al. [14] who first introduced RAG with dense retrieval, leveraging the success of Dense Passage Retrieval [18] to improve knowledge-intensive QA.

3.3.2 Sparse Retrieval RAG

The sparse retrieval RAG system replaces the dense vector search with a BM25-based lexical search. In many ways, its implementation is similar to the dense pipeline, but with the embedding and vector store steps removed and a different retriever in place. Here's how the sparse RAG pipeline operates:

1. **Document Loading and Chunking:** We begin exactly as before by loading all documents from files, using `DirectoryLoader` and `TextLoader`, and splitting them into chunks with `CharacterTextSplitter`. At this stage, we have our list of chunked `Document` objects, identical to what we had in the dense setup. The chunking strategy remains the same, ensuring that chunks are reasonably sized and self-contained. It is important to note that chunking benefits the sparse retriever as well: shorter chunks mean the query's terms either appear or not in a chunk and we do not dilute the BM25 score across a very long document. It also means if only one section of a long document is relevant, we isolate that section as a chunk which can be retrieved on its own.

2. **BM25 Indexing:** Instead of embedding the chunks, we start by initializing the `BM25Retriever` with the list of chunks. For example, in code we call something like:

   ```
   sparse_retriever = BM25Retriever.from_documents(chunks).
   ```

   This call builds an internal inverted index of terms to chunk IDs and computes the necessary IDF statistics for BM25. The retriever uses a library `rank_bm25` under the hood to support BM25 scoring. There is no external store or database needed, but only an in-memory index since our corpus is of a manageable size.

3. **Query Retrieval with BM25:** When a user question is asked, we use `sparse_retriever.get_relevant_documents(query)`. The BM25 retriever will tokenize the query, while applying simple text preprocessing and then compute a BM25 score for each chunk's text. BM25 scoring formula takes

into account term frequency in the chunk, chunk length and term rarity across the corpus. The retriever returns the top k chunks with highest scores. For instance, if the query is "What is the capital of Australia?", the BM25 retriever will look for chunks containing words like "capital" and "Australia" and rank them. A chunk that says "... Canberra is the capital of Australia..." would score very high due to containing both query terms, whereas a chunk about Australian wildlife with no mention of the word capital would score zero or very low. The number of documents to retrieve of k is set similarly to the dense case to ensure multiple pieces of context in case some are only partially relevant.

4. **Prompt Assembly:** The retrieved chunks from BM25 in the form of Document objects with text are then formatted into the prompt using the same `ChatPromptTemplate` as before. We fill in the context section with the texts of these top chunks. In our notebooks, this was done identically to the dense pipeline; the difference is purely that the source of these chunks is BM25 rather than vector search. In code, one could reuse the same prompt template object and just supply it new context. The question itself is the same user query.

5. **LLM Generation and Output:** We call the `ChatOpenAI` model with the prompt: system instructions, user context and question and parse the output with `StrOutputParser`. The LLM will generate an answer based on the context provided. Because the context came from BM25, which tends to return documents containing literal query terms, the model will often see very directly relevant text. For factoid questions, this can be very effective: the answer sentence might be present exactly among the chunks.

3.3.3 Hybrid Retrieval RAG

The hybrid RAG system integrates both dense and sparse retrieval to provide the LLM with a more comprehensive set of context documents. Its implementation builds on parts of the previous two pipelines and adds an ensemble mechanism. The steps for the hybrid pipeline are:

1. **Document Loading and Chunking:** Just as with the other variants, we start by loading all documents and splitting them into chunks.

2. **Initialize Dense and Sparse Retrievers:** We set up both retrieval mechanisms in parallel. First, we create the dense retriever: embed all chunks using `OpenAIEmbeddings` and store them in a `Chroma` vector store (same as in §3.3.1). Then we call `vector_store.as_retriever()` to get a dense retriever

object. Separately, we initialize the BM25 retriever by

`BM25Retriever.from_documents(chunks)`.

3. **Ensemble Setup (Reciprocal Rank Fusion):** We create an `EnsembleRetriever` object with the two retrievers:

   `retrievers=[dense_retriever, sparse_retriever], weights=[1,1]`

   `hybrid_retriever = EnsembleRetriever(retrievers)`.

   At query time, it calls both retrievers, applies Reciprocal Rank Fusion [17] to merge their results and returns a unified ranked list.

4. **Query Retrieval (Hybrid):** We then call

   `hybrid_retriever.get_relevant_documents(query)`.

   The result is a fused set of top-k chunks that reflect both semantic and lexical relevance.

5. **Prompt and Generation:** The fused chunks are injected into the prompt as context. Then, the query fills the question slot and the ChatOpenAI model generates the answer. Lastly, StrOutputParser extracts the final text.

The hybrid RAG aims to inherit the strengths of both dense and sparse retrieval. Dense retrieval captures semantic similarity and robustness to paraphrasing; BM25 ensures that exact matches on critical tokens are not missed. Prior work on hybrid retrieval confirms that such combinations often achieve higher recall and improved QA performance compared to single-method systems [19, 17]. The trade-off is a modest increase in computational and implementation complexity. Nonetheless, this hybrid system forms a strong baseline for our subsequent experiments, against which the added value of GraphRAG can be meaningfully assessed.

3.4 GraphRAG

GraphRAG is a retrieval-augmented generation pipeline that extends the conventional RAG approach to handle broad, sense-making queries over an entire corpus[10]. Unlike a standard RAG system, which typically retrieves only a small set of semantically similar passages per query, GraphRAG constructs a knowledge graph over the corpus and employs hierarchical summarisation to synthesise answers that reflect a more global view of the underlying data. The pipeline consists of several offline stages, which are knowledge graph construction, graph-based community detection and community-level summarisation, followed by an online query

stage in which a map-reduce style procedure combines the precomputed summaries into a final answer. The central idea is to decompose the corpus into meaningful components via the graph and its communities, summarise each component and then recombine those summaries for question answering.

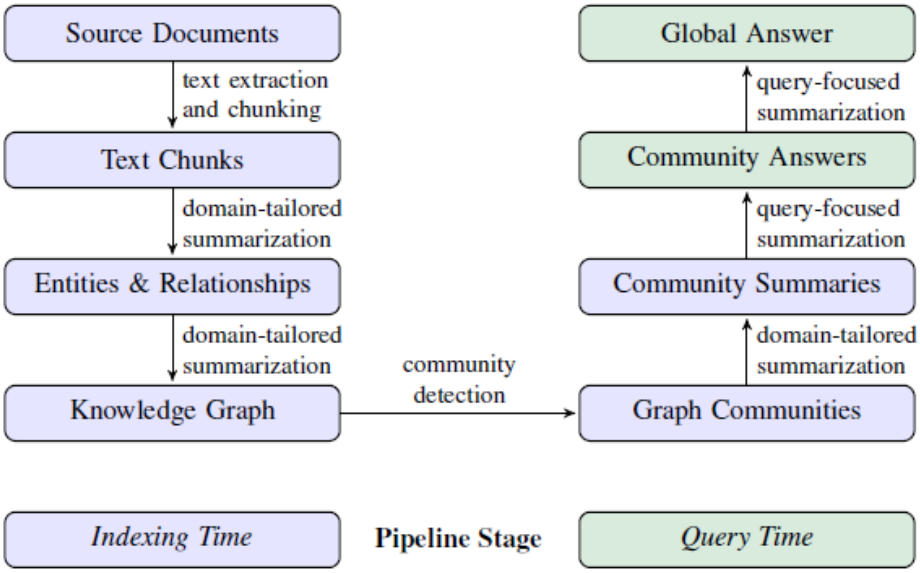

Figure 3.1: High-level architecture of the GraphRAG pipeline.

3.4.1 Knowledge Graph Construction

Document chunking and extraction. The process begins by splitting source documents into smaller chunks, ensuring that each unit can be reliably processed by the LLM. For each chunk, the LLM is prompted to extract a set of structured elements. First, salient *entities* such as people, organizations, locations and technical concepts are extracted. Then, *relationships* between these entities are extracted. Lastly, it extracts relevant factual *claims* or *attributes* associated with the entities. For example, a chunk describing an acquisition would yield entities for the companies involved, a relationship encoding the acquisition event and claims specifying properties such as the year or value. Each extracted entity, relation and claim acts as a condensed representation of the information present in that chunk.

Graph assembly. Once all the chuncks have been squeezed out, all extracted elements are then merged into a single knowledge graph. Each unique entity corresponds to a node and relationships become edges connecting the respective

nodes. When the same entity or relation appears multiple times in different sections, the occurrences are consolidated. Entity mentions are merged into a single node, and their descriptions and metadata are aggregated. Repeated relations form a single edge, and the edge's weight can reflect the frequency or strength of the connection. In our implementation, a simple string-based normalisation is applied to merge identical entity names; unresolved variants are typically captured together in later community detection. The resulting graph serves as a structured index of the corpus in which nodes carry LLM-generated descriptions, edges encode how entities are related optionally with associated claims and the overall structure captures how concepts in the documents are interconnected.

3.4.2 Graph Community Detection

Once the knowledge graph is constructed, GraphRAG partitions it into groups of closely related entities using graph community detection. The aim is to identify communities in which nodes are densely connected to each other and more sparsely connected to the rest of the graph, so that each community corresponds to a coherent subtopic or theme.

To obtain such partitions, we employ the Leiden algorithm [11], an improvement over the Louvain method that yields well-connected and stable communities. Leiden is first applied to the full graph to obtain an initial partition. Then, the same procedure is recursively applied to the induced subgraph of each community, producing a hierarchy of communities at multiple levels of granularity. At any given level, the communities form a partition of the node set, with lower levels producing small, fine-grained clusters and higher levels merging related clusters into broader thematic groups.

This hierarchical clustering is essential for GraphRAG's divide-and-conquer strategy. By decomposing the knowledge graph into thematically coherent communities, we obtain smaller units that can be processed and summarised independently. Intuitively, each community captures a segment of the corpus focused on a particular topic or set of closely linked entities, enabling later stages to reason locally within communities while still supporting a global view through the hierarchy.

3.4.3 Hierarchical Community Summarisation

For each community in the hierarchy, GraphRAG generates a concise textual summary. These *community summaries* describe the main entities in the community, the most important relationships between them and key facts or events. By associating a summary with every community, at every level, the system builds a multi-scale, human-readable index of the corpus.

Summarisation is performed bottom-up. For a leaf-level community, which is a

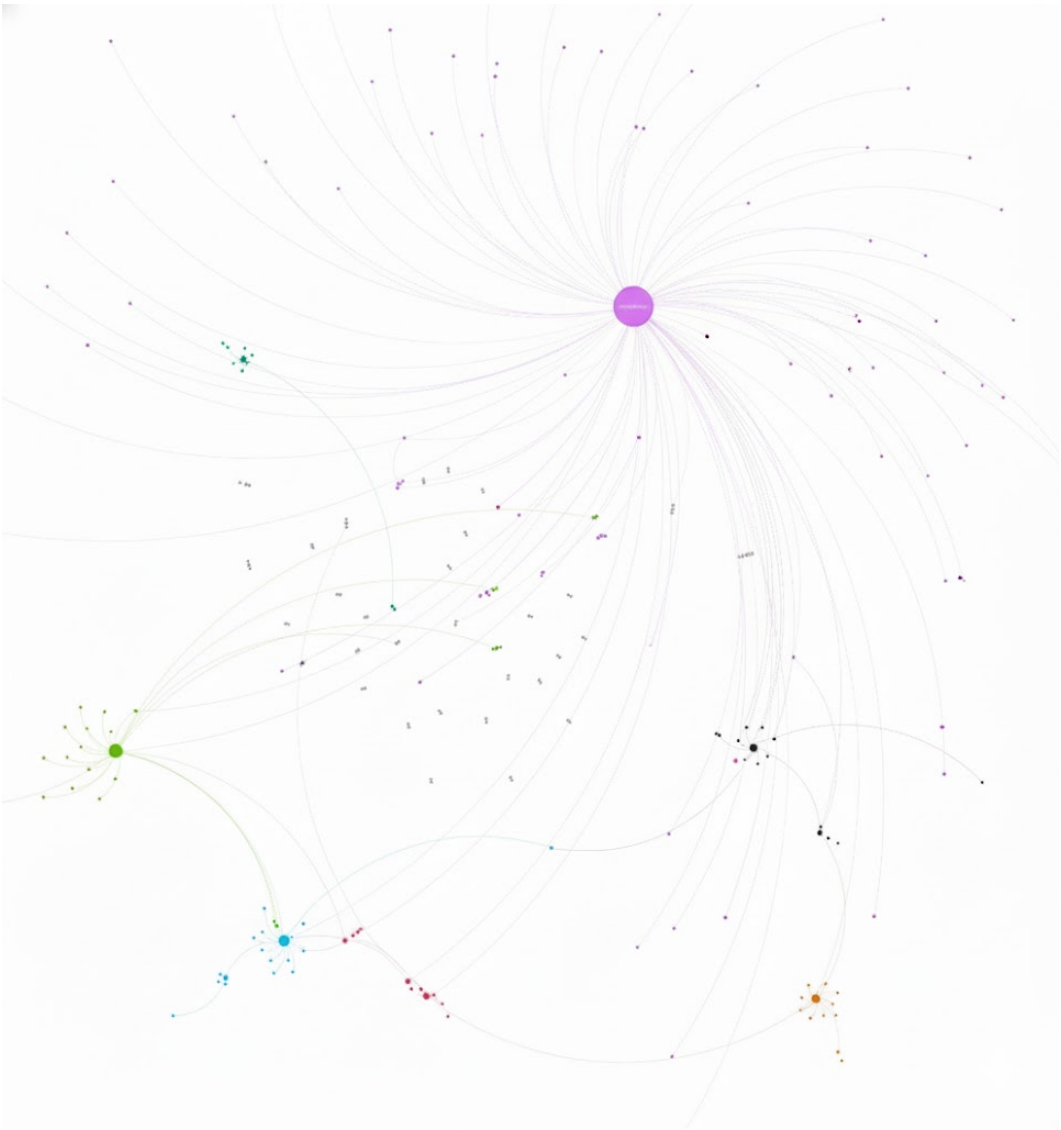

Figure 3.2: Community-structured graph of Oppenheimer's Wikipedia page, extracted via GraphRAG; each color represents a cohesive cluster.

small, focused cluster, the system constructs a prompt containing: selected entity descriptions, their most important relations, and any high-value claims within that subgraph. The LLM is instructed to synthesise these into a short, coherent summary capturing what that community is "about".

For higher-level communities, which aggregate multi sub-communities, a recursive strategy is adopted by graphRAG. Rather than inlining all low-level details, which would exceed the context window, the prompt for a higher-level community is

organised including a subset of particularly central entities and relations at that level and the existing summaries of some of its child communities. In this way, detailed information is compressed into lower-level summaries and higher-level prompts operate on these condensed representations. The LLM then produces a higher-level summary that integrates the content of its children. Repeating this process up the hierarchy yields one or a small number of global summaries that provide a bird's-eye overview of the entire corpus.

The outcome is a hierarchy of precomputed summaries: from fine-grained descriptions of small clusters to broad overviews of major themes. These summaries are later reused as structured context for answering user queries.

3.4.4 Query-Time Answer Synthesis

At query time, GraphRAG exploits the hierarchy of community summaries to construct answers via a map-reduce style procedure[10]. This allows the system to incorporate information from a large portion of the corpus without exceeding the LLM's context constraints.

Selecting summary context. Given a user question, the system selects an appropriate level of the hierarchy: higher levels are used for very broad questions, whereas lower levels can be chosen for more specific queries. All summaries at the chosen level are treated as candidate context and split into multiple chunks so that each chunk fits within the model's input window.

Map stage. For each chunk of summaries, the system queries the LLM with the user's question and that chunk as context. The model is asked to produce a provisional answer based only on the provided summaries and estimate how useful this answer and the underlying chunk is for the question. Chunks that yield uninformative or irrelevant answers can be discarded based on this score. This step effectively "maps" different parts of the summary space to candidate contributions to the final answer.

Reduce stage. The remaining provisional answers are ranked by their usefulness scores. The most informative ones are concatenated, up to a certain token budget, into a new aggregated context. Finally, the LLM is prompted once more with the original question and this aggregated context to produce a single, coherent final answer. In this reduce step, the model performs a focused synthesis over the most relevant partial answers, combining evidence sourced from multiple communities.

Through this process, GraphRAG enables information distributed across many documents and topics to influence the final response in a controlled and interpretable way. The knowledge graph provides structure, the Leiden-based hierarchy organises

that structure into themes, the community summaries distil each theme and the map-reduce query procedure recombines those distilled units to answer complex questions with a more global perspective than conventional vector-based RAG can typically offer.

3.4.5 Implementation Details and Extensions

Our work builds upon the open-source GraphRAG implementation released by Edge et al. [10]. While the original project organises the code into `data/`, `graphrag/` and `scripts/`, our contributions focus on providing a streamlined entry point for constructing and querying GraphRAG in practical scenarios. In particular, we initially encapsulate the full pipeline into a single executable script and expand data ingestion to include Neo4j graph databases. Then we exposed a simple interface for custom Cypher queries and prompt engineering and integrate the RAGAS evaluation framework natively into the workflow.

Simplified pipeline. The core script in the expansion bundles together document ingestion, entity extraction, graph construction, community detection and summarisation into a cohesive pipeline. It reads configuration parameters from a single YAML file, such as chunk sizes, overlap, model names and orchestrates the stages described earlier. This design shields the user from the complexity of manually running separate scripts and enables rapid experimentation: one can, for example, adjust the chunk size or the number of top relations included in summaries by editing the configuration rather than modifying multiple files.

Neo4j data ingestion. A notable extension is support for ingesting existing knowledge from a Neo4j database. Through a dedicated module (`generate_reports.py`) we connect to a running Neo4j instance, extract nodes, relationships and their properties via Cypher queries and convert them into the format expected by the GraphRAG pipeline. This allows users to enrich the knowledge graph with curated information, like organisational charts or domain ontologies, before community detection and summarisation. Neo4j ingestion complements the standard document-based extraction: the pipeline can merge entities from the Neo4j graph with those extracted from text, thereby unifying structured and unstructured sources.

Custom Cypher queries and prompts. To facilitate advanced exploration of the graph, we expose a simple interface for running Cypher queries directly from within the GraphRAG environment. Users can compose queries to fetch subgraphs or specific patterns, such as “find all projects connected to a given department”, and then feed the resulting nodes into the local retrieval module.

In addition, we introduce a mechanism to insert or override prompt templates on a per-query basis. This is useful when one wants to tailor the instructions given to the LLM for a particular use case, without changing the global configuration. For instance, For exploratory analysis or debugging, a concise prompt can be used to request short, structured outputs (e.g. bullet points or JSON) so that graph content, communities or Cypher query results can be inspected quickly without verbose prose. In contrast, when the goal is to produce documentation-like answers, a different prompt can require references to specific nodes, communities or source passages, encouraging the model to expose its evidence and making the behaviour of GraphRAG more transparent. In an enterprise setting, per-query prompts also allow one to enforce style or policy constraints, such as avoiding speculative statements or enforcing domain-specific terminology, without modifying the underlying pipeline. Overall, the ability to plug in query-specific instructions is a practical wrapper around GraphRAG, adapting the same graph-based index to different use cases and audiences while keeping the implementation unchanged.

RAGAS integration. Finally, we embed the RAGAS evaluation framework into the pipeline. After generating an answer, the system automatically computes metrics such as context recall, context precision, answer relevance and answer faithfulness. The evaluation results are logged alongside the answer and can be aggregated to compare different parameter settings or retrieval modes. This built-in evaluation loop enables systematic analysis of how pipeline choices affect answer quality.

In summary, our implementation preserves the conceptual foundations of the knowledge graph construction of GraphRAG, community detection with Leiden clustering and hierarchical summarisation—while adding practical enhancements aimed at simplifying use and broadening applicability. By supporting Neo4j ingestion, custom Cypher queries, flexible prompt design and automatic RAGAS scoring, our implementation makes GraphRAG accessible for enterprise scenarios where both structured databases and document corpora must be integrated and evaluated.

Chapter 4

Metrics and Dataset

This chapter introduces the evaluation tools and datasets used to measure the performance of our retrieval-augmented generation (RAG) pipelines. We first present the **RAGAs** framework, which provides a set of reference-free, LLM-based metrics designed to jointly assess retrieval quality, answer faithfulness, and alignment between model outputs and their supporting evidence. These metrics allow us to go beyond surface-level similarity scores and explicitly diagnose whether errors stem from inadequate retrieval, hallucinations, or poor use of context. We then describe the two benchmarks underpinning our experiments: **WikiEval**, a recent dataset tailored for evaluating RAG systems on post-2022 Wikipedia content, and a carefully constructed subset of **HotpotQA** focused on multi-hop reasoning. Together, these resources enable a systematic and controlled comparison of different RAG architectures across single-hop, noisy-context, and multi-hop scenarios, ensuring that our evaluation is both realistic and aligned with the challenges faced in enterprise settings.

4.1 RAGAs Evaluation Framework

Evaluating retrieval-augmented generation (RAG) systems poses significant challenges: one must assess both the quality of the retrieved evidence and the quality of the answer produced. In natural language processing, traditional metrics either gauge answer similarity (e.g., BLEU, ROUGE) or retrieval performance (e.g., recall and precision) in isolation. They do not capture whether an answer truly reflects the information retrieved or if the retrieved passages actually cover the needed content. Recent work underscores the need for holistic metrics that jointly evaluate retrieval and generation. An ideal assessment should determine:

- **Retrieval adequacy:** Does the retriever find enough relevant information to address the query?

- **Faithful use of context:** Does the language model's output faithfully use that information?
- **Answer quality:** Is the final answer both correct and clear?

Several benchmarks attempt to capture these aspects. For example, the publicly released WikiEval dataset pairs questions drawn from fifty post-2022 Wikipedia pages with their ideal context and answers so that researchers can evaluate both retrieval quality and generative accuracy [20]. However, most domains lack such curated question–answer sets, which makes evaluation challenging. To address this gap, **RAGAs** (Retrieval-Augmented Generation Assessment) provides a reference-free evaluation framework that scores RAG pipelines without needing gold answers [13]. RAGAs applies a suite of metrics to gauge how well a system retrieves and uses information. We summarize its four core measures below and explain why we adopt them in this thesis.

Answer Relevancy. This metric checks whether the generated answer directly addresses the user's question. RAGAs prompts an LLM to "reverse-engineer" several questions from the answer and measures the cosine similarity between these generated questions and the original query. If the answer is on-topic and complete, the reconstructed questions will be similar to the original; if it drifts off topic or omits key details, the similarity drops. In practice, answer relevancy highlights responses that are irrelevant or incomplete, regardless of factual correctness [13].

Answer Faithfulness. Faithfulness measures whether the answer's claims are grounded in the retrieved context. RAGAs first asks an LLM to break the answer into individual factual statements, then asks another LLM to verify whether each statement can be inferred from the retrieved documents. The score is the fraction of statements deemed supported by the context. An answer that introduces facts not present in the sources will score lower, signaling hallucinations or unsupported assertions [13].

Contextual Precision. Precision evaluates the retrieval step by checking how much of the fetched context is actually relevant. Formally, it computes the proportion of context chunks that contain useful information for answering the question. High precision means the retriever supplied mostly pertinent passages; low precision indicates that many retrieved documents are extraneous or noisy [13].

Contextual Recall. This metric complements precision by measuring coverage: it assesses how many of the answer's factual claims can be supported by the retrieved context. The score is the fraction of answer statements that are found in

the retrieved documents. A high recall suggests that the retriever captured most of the necessary evidence, whereas a low recall points to missing information [13].

Why RAGAs in this Thesis? We adopt RAGAs because it provides a holistic, reference-free way to evaluate our enterprise RAG pipeline. The combination of answer relevancy and faithfulness allows us to check that the model not only stays on topic but also grounds its statements in the sources. Contextual precision and recall quantify how well our retrieval module filters and covers the information needed to answer each query. Together, these metrics enable us to diagnose whether errors stem from poor retrieval or from misusing context, guiding improvements to both components. Moreover, because RAGAs does not require human-written reference answers, it scales to our setting, where we test on publicly available datasets like WikiEval and later apply the same methodology to proprietary enterprise documents [13].

4.2 The WikiEval Dataset

4.2.1 Origin and Construction of the Dataset

WikiEval is a dataset specifically designed to evaluate context-aware question answering systems, such as Retrieval-Augmented Generation (RAG) pipelines. Due to the lack of publicly available datasets with pre-written questions, context passages and annotated answers, the authors of [13] constructed a new benchmark. They selected 50 Wikipedia pages on recent topics (post-2022) and extracted their introductory sections. Using these passages, ChatGPT was prompted to create a question of moderate difficulty that could be fully answered using only the provided context.

For each question, three types of answers were generated: a *grounded* answer based on the reference context, an *ungrounded* answer produced without context, potentially less accurate, and a deliberately *poor* answer. Likewise, the context was diversified into two variants: a concise version (`context_v1`) and a noisy version (`context_v2`) enriched with tangential information.

Each record in the dataset includes the question, the Wikipedia source, all three answer variants and both context types. Two human annotators independently rated each item along three axes: faithfulness to the context, answer relevance and context relevance—reaching over 95% agreement on faithfulness and about 90% on answer relevance. These annotations serve as human-validated ground truth for evaluating automated metrics.

4.2.2 Advantages of using WikiEval in Our Experiments

WikiEval was selected for our experiments for several compelling reasons. First, it is openly available via Hugging Face [20], allowing seamless integration without needing to build a dataset from scratch.

Second, WikiEval provides a balanced variety of factual and complex queries. For instance, a simple factual example is:

> *"When is the scheduled launch date and time for the PSLV-C56 mission and where will it be launched from?"*

which can be resolved through direct extraction. In contrast, a more elaborate question is:

> *"What is the objective of the Uzbekistan-Afghanistan-Pakistan Railway Project and how is it expected to enhance trade and logistics efficiency?"*

which requires multi-sentence reasoning and synthesis.

Finally, WikiEval was used in the original RAGAS paper to validate its automated evaluation metrics against human judgment [13, 12]. This ensures compatibility with our chosen evaluation framework and confidence in the alignment between human and metric-based assessments. Its structure and annotations make it a reliable choice for testing our RAG system's performance across answer quality and context faithfulness.

4.3 The HotpotQA Dataset

HotpotQA is a large-scale question answering benchmark specifically designed to evaluate multi-hop reasoning over Wikipedia articles [21]. It contains about 113,000 question–answer pairs where each question is constructed to require information from *at least* two distinct pieces of evidence, typically drawn from different Wikipedia pages. In contrast to earlier single-hop or span-matching datasets, HotpotQA explicitly targets scenarios where a system must combine multiple clues, follow links between entities, or compare properties across entities in order to arrive at the correct answer.

The dataset is built through a structured crowdsourcing pipeline. First, the authors construct a hyperlink graph over Wikipedia, focusing primarily on the first paragraphs of articles where key entities and relations are concentrated. Candidate pairs of pages are then sampled from this graph to encourage meaningful connections between entities; for example, an entity and its related concept or two entities that share a relation. Crowd workers are presented with these candidate page pairs and asked to write questions whose solution requires using information from both pages.

This procedure yields natural language questions whose supporting evidence is distributed across multiple documents rather than confined to a single paragraph.

A central strength of HotpotQA is that, in addition to providing the answer span, it offers supporting facts at the sentence level. Some annotators mark which sentences on which pages are necessary to answer each question. These labels make it possible to assess not only whether a model predicts the correct answer, but also whether it can recover the underlying reasoning chain. The dataset also introduces two key categories of multi-hop questions:

- **Bridge questions**: where the model must infer or follow an intermediate entity (a "bridge") from one page to another, such as identifying a person or place in the first article and then using that entity to retrieve information from a second article.
- **Comparison questions**: where the model must retrieve facts about two entities and explicitly compare them before selecting the correct answer. For example, answering a question may require storing dates, numerical values, or categorical attributes for comparison.

These design choices make HotpotQA particularly suitable for evaluating systems that claim to perform multi-hop reasoning: solving a question typically requires aggregating distributed evidence, resisting shortcuts based only on lexical overlap, and, for comparison questions, executing simple but explicit reasoning over retrieved facts. As a result, HotpotQA has become a standard benchmark for multi-hop QA and for retrieval-augmented and iterative reasoning methods.

4.3.1 Subset Construction for This Thesis

Since HotpotQA would require chunking and embedding around 100,000 Wikipedia pages, it is not used at full scale in this thesis. Instead, it is used as a controlled benchmark to analyze the behavior of different RAG architectures under multi-hop conditions. We construct a focused subset of the dataset with the following procedure:

1. We select **100 questions** from the original HotpotQA corpus.
2. For each question, we retain the two associated supporting Wikipedia pages specified by the dataset annotations, obtaining a corpus of **200 Wikipedia articles** in total. This preserves the original multi-hop structure while keeping the document collection compact and fully observable.
3. To cover both main reasoning patterns defined in HotpotQA, we enforce a balanced split: **50 bridge** questions and **50 comparison** questions.

4. Within each of these two groups, we stratify by difficulty according to the dataset metadata:
    - 20% *easy*,
    - 50% *medium*,
    - 30% *hard*.

This construction yields a subset that is small enough for detailed analysis, yet rich enough to stress the capabilities of the examined systems. The balanced design ensures that:

- Both bridge-style questions, which test the ability to follow multi-step entity links and intermediate facts, and comparison-style questions, which test aggregation and comparison over multiple entities, are equally represented.
- Each RAG variant is evaluated across a spectrum of difficulties, from straightforward retrieval-and-lookup to more challenging cases requiring careful retrieval, disambiguation, and composition of evidence.

By grounding our experiments in this curated HotpotQA subset, we can directly assess how dense, sparse, hybrid, and graph-based RAG systems cope with genuine multi-hop reasoning requirements, without the confounding factors of an excessively large or noisy corpus.

Chapter 5

Results

5.1 Results and Discussion

5.1.1 Experimental Setup

We evaluate five retrieval pipelines: three *baseline* RAG systems (**Sparse**, **Dense**, **Hybrid**) and two *GraphRAG* modes (**Local**, **Drift**).[1] All systems use the same LLM configuration, `gpt-4o-mini`, temperature 0 and `text-embedding-3-small` for the baeline RAG, and the two datasets: the WikiEval dataset and the HotpotQA subset, described in Section 4.2 and in Section 4.3.1. We report results for two prompt variants (Section 5.1.3), and for GraphRAG we additionally sweep the *community level* from 0 to 3 (Section 5.1.4).

5.1.2 Metrics (RAGAs)

We adopt the RAGAs framework to quantify: *Answer Faithfulness*, *Answer Relevancy*, *Context Precision*, and *Context Recall*. Faithfulness captures whether the final answer is supported by the retrieved context; Relevancy measures the answer–question alignment; Precision rewards focused retrieval; Recall captures coverage of the necessary evidence. Scores are reported as mean ± 95% confidence interval across questions. For what it concerns the HotpotQA dataset, we also provide stratified results by *difficulty*: easy, medium and hard and *question type*: bridge and comparison.

[1] Drift: denotes the global mode in which community summaries are used as primary context.

5.1.3 Prompt Variants

Each pipeline is evaluated under two prompts:

($\mathbf{P_{simple}}$) Minimal prompt.

```
Question: {question}
Context: {context}
Answer:
```

($\mathbf{P_{grounded}}$) Grounded, concise prompt.

```
You are an assistant for question-answering tasks.
Use the following pieces of retrieved context to answer the question.
(The exact answer is less than 5 words.) \text{--only for hotpotQA--
If you don't know the answer, just say that you don't know.
Do not add information from outside the documents.
Question: {question}
Context: {context}
Answer:
```

The grounded prompt enforces conservative, short answers and explicit grounding, which typically improves *Faithfulness* and *Precision*. For GraphRAG, we insert the $P_{grounded}$ prompt bothin the map and reduce prompts, in the part where it describes the type of response.

5.1.4 GraphRAG Community Levels

For GraphRAG, we vary the *community level* used to assemble context and/or summaries: **0, 1, 2, 3**. Level 3 uses the most granular communities (leaf level), while higher levels aggregate broader clusters. This sweep tests the trade-off between *detail* (lower levels) and *coverage/compression* (higher levels).

5.1.5 Evaluation Protocol

For each system and prompt:

- Run inference on the 50 WikiEval Questions
- Run inference on the 100 HotpotQA questions.
- Compute RAGAs metrics for each question.
- Aggregate the results overall and by slices of difficulty and question type.

- For *GraphRAG (Local/Drift)*, repeat evaluation for community levels 0–3 and select the best-performing level according to the macro-average of *Faithfulness* and *Relevancy*.

5.1.6 Main Results

WikiEval dataset. Table 5.1 summarises the RAGAs metrics for each system on the WikiEval dataset using both prompt variants. Results are reported as mean ± 95% CI.

Table 5.1: RAGAs metrics on the WikiEval dataset (mean ± 95% CI). P_{simple} vs. $P_{grounded}$. Best per column in **bold**.

System (Prompt)	Faithfulness	Relevancy	Ctx Precision	Ctx Recall
Sparse (P_{simple})	0.937 ± 0.006	0.949 ± 0.015	0.751 ± 0.042	0.938 ± 0.011
Sparse ($P_{grounded}$)	0.953 ± 0.008	0.952 ± 0.017	0.731 ± 0.051	0.956 ± 0.017
Dense (P_{simple})	0.976 ± 0.014	0.945 ± 0.015	$\mathbf{0.915 \pm 0.040}$	0.947 ± 0.008
Dense ($P_{grounded}$)	0.985 ± 0.006	0.956 ± 0.017	0.905 ± 0.042	0.940 ± 0.011
Hybrid (P_{simple})	0.970 ± 0.012	0.949 ± 0.013	0.819 ± 0.026	0.938 ± 0.011
Hybrid ($P_{grounded}$)	0.954 ± 0.017	$\mathbf{0.989 \pm 0.010}$	0.824 ± 0.046	0.950 ± 0.010
G-RAG L. (P_{simple})	0.979 ± 0.008	0.966 ± 0.014	0.878 ± 0.033	0.952 ± 0.017
G-RAG L. ($P_{grounded}$)	0.987 ± 0.010	0.971 ± 0.011	0.892 ± 0.030	0.958 ± 0.023
G-RAG D. (P_{simple})	0.983 ± 0.008	0.964 ± 0.010	0.880 ± 0.034	0.956 ± 0.014
G-RAG D. ($P_{grounded}$)	$\mathbf{0.991 \pm 0.007}$	0.970 ± 0.012	0.888 ± 0.031	$\mathbf{0.961 \pm 0.012}$

Interestingly, the data shows that Local GraphRAG slightly outperforms its Drift implementation in this dataset. This is not surprising, since this dataset contains 50 questions that do not require intensive comparison of Wikipedia pages or global reasoning. Therefore, a local GraphRAG is better suited for these tasks, and since it is less computationally expensive, it is clearly a more suitable model.

The other data are as expected. We can see that the Hybrid Rag has better overall results compared to the other classic baselines. The Dense retrieval results are comparable to the Sparse ones, with a very slight advantage for the former, because of the presence of some broader question such as *"What is the objective of the Uzbekistan-Afghanistan-Pakistan Railway Project and how is it expected to enhance trade and logistics efficiency?"* and *"What factors contributed to the Sri Lankan economic crisis?*.

For the models tuned with the $P_{grounded}$ we observe an improvement for the *Context Precision* and *Faithfulness*.

HotpotQA subset. Table 5.2 reports the same metrics on the HotpotQA subset described earlier. As expected, the requirement of more complex multi-hop reasoning emphasizes the gap between GraphRAG and classic models.

In this setting, the Drift mode clearly outperforms the Local mode, underscoring GraphRAG's advantage on complex, sense-making tasks such as comparisons and cross-entity linking. For similar reasons, the gap between the classical RAG baselines and GraphRAG also widens. The sparse (BM25) retriever is particularly disadvantaged: it relies on lexical matching and term proximity, whereas in this case the salient terms are dispersed across passages and often paraphrased, which undermines exact-match scoring and hampers retrieval of the necessary evidence.

Table 5.2: RAGAs metrics on the HotpotQA subset (mean ± 95% CI). P_{simple} vs. $P_{grounded}$. Best per column in **bold**.

Sparse (P_{simple})	0.560 ± 0.095	0.655 ± 0.063	0.312 ± 0.078	0.720 ± 0.095
Sparse ($P_{grounded}$)	0.771 ± 0.044	0.908 ± 0.013	0.305 ± 0.078	0.720 ± 0.090
Dense (P_{simple})	0.575 ± 0.091	0.736 ± 0.052	0.402 ± 0.099	0.750 ± 0.066
Dense ($P_{grounded}$)	0.809 ± 0.034	0.929 ± 0.019	0.397 ± 0.098	0.750 ± 0.070
Hybrid (P_{simple})	0.696 ± 0.046	0.927 ± 0.025	0.384 ± 0.099	0.580 ± 0.100
Hybrid ($P_{grounded}$)	0.696 ± 0.046	0.927 ± 0.025	0.384 ± 0.099	0.580 ± 0.100
G-RAG L. (P_{simple})	0.865 ± 0.028	0.945 ± 0.018	0.420 ± 0.062	0.795 ± 0.048
G-RAG L. ($P_{grounded}$)	0.902 ± 0.024	0.952 ± 0.016	$\mathbf{0.448 \pm 0.058}$	0.812 ± 0.045
G-RAG D. (P_{simple})	0.904 ± 0.022	0.948 ± 0.017	0.404 ± 0.060	0.835 ± 0.042
G-RAG D. ($P_{grounded}$)	$\mathbf{0.938 \pm 0.019}$	$\mathbf{0.957 \pm 0.015}$	0.426 ± 0.057	$\mathbf{0.862 \pm 0.040}$

HotpotQA: difficulty breakdown. Table 5.3 summarises performance on easy vs. medium vs. hard questions in the HotpotQA subset. We typically see larger gains from *GraphRAG (Drift)*, because of more multi-hop and reasoning load. On the other side, for "easy" and "medium" questions the distance is reduced.

Table 5.3: RAGAs metrics on HotpotQA by difficulty (easy / medium / hard).

System	Faithfulness	Relevancy	Ctx Precision	Ctx Recall
Sparse (e./m./h.)	*0.81/0.84/0.67*	*0.86/0.84/0.94*	*0.32/0.34/0.25*	*0.88/0.84/0.67*
Dense (e./m./h.)	*0.86/0.81/0.81*	*0.90/0.93/0.92*	*0.46/0.49/0.25*	*0.68/0.60/0.85*
Hybrid (e./m./h.)	*0.76/0.78/0.64*	*0.94/0.94/0.94*	*0.57/0.49/0.23*	*0.66/0.63/0.50*
G-RAG L.(e./m./h.)	*0.90/0.91/0.86*	*0.95/0.96/0.93*	*0.47/0.45/0.39*	*0.82/0.81/0.80*
G-RAG D.(e./m./h.)	*0.92/0.95/0.90*	*0.96/0.97/0.95*	*0.45/0.43/0.36*	*0.86/0.88/0.84*

HotpotQA: question type breakdown. Table 5.4 reports results on the HotpotQA subset by question type *bridge vs. comparison*. Both bridge and comparison questions benefits from global summaries *GraphRAG Drift*. Questions benefit from P_{grounded} due to concise formatting, so we decided to evaluate with this prompt. All classical baselines in general struggle with the HotpotQA dataset.

Table 5.4: RAGAs metrics on HotpotQA by question type (bridge / comparison).

System	Faithfulness	Relevancy	Ctx Precision	Ctx Recall
Sparse (bridge / comp.)	*0.81/0.74*	*0.90/0.92*	*0.34/0.27*	*0.80/0.64*
Dense (bridge / comp.)	*0.78/0.84*	*0.91/0.95*	*0.39/0.40*	*0.64/0.76*
Hybrid (bridge / comp.)	*0.76/0.64*	*0.93/0.95*	*0.37/0.40*	*0.40/0.76*
G-RAG Local (bridge / comp.)	*0.91/0.89*	*0.96/0.97*	*0.58/0.61*	*0.84/0.80*
G-RAG Drift (bridge / comp.)	*0.94/0.91*	*0.97/0.97*	*0.57/0.60*	*0.88/0.83*

5.1.7 Effect of Prompt Tuning

Across all pipelines and datasets, P_{grounded} increases *Faithfulness* and *Ctx Precision* with neutral-to-positive effects on *Relevancy*. Especially in the hotpotQA the effect of suggesting to use *"less than 5 words"* is positive and it helps to reduce speculation and verbosity. Gains are often most pronounced for Dense and Hybrid systems, and remain positive for GraphRAG.

5.1.8 GraphRAG Community Level Sweep

To assess the effect of community granularity on GraphRAG performance, we sweep *community levels* 0–3 for both **Local** and **Drift** modes. Table 5.5 reports mean RAGAs scores aggregated across both datasets. The values are RAGAs scores mean ± 95% CI aggregated across WikiEval and HotpotQA. For these experiments, we used the P_{grounded} model. In general, intermediate levels (1–2) offer the best balance between detail and coverage, while level 3, which is composed by the leaf-level communities, maximises detail but can reduce *Context Precision* and it does not show progress in any metrics. On the other side level 0 risks diluting key evidence in broad summaries, which is especially evident in the hotspotQA dataset.

5.1.9 Discussion

The results across WikiEval and HotpotQA reveal several insights:

- **GraphRAG Drift** excels on multi-hop and globally scoped queries due to its use of summarised communities, improving Faithfulness and Recall on

Table 5.5: Community level sweep for GraphRAG (**Local** and **Drift**).

. The value are presented in the format wikiEval/HotpotQA

Mode	Level	Faithfulness	Relevancy	Ctx Precision	Ctx Recall
Local	0	0.982/0.888	0.968/0.948	0.880/0.440	0.952/0.800
Local	1	0.987/0.906	0.972/0.953	**0.892**/**0.452**	0.958/0.816
Local	2	0.986/0.904	**0.973**/0.954	0.890/0.448	0.957/0.814
Local	3	0.983/0.898	0.970/0.950	0.878/0.438	0.955/0.808
Drift	0	0.985/0.920	0.969/0.952	0.882/0.430	0.958/0.848
Drift	1	0.990/0.936	0.971/0.956	0.888/0.424	0.960/0.860
Drift	2	**0.991**/**0.940**	0.972/**0.957**	0.887/0.422	**0.961**/**0.864**
Drift	3	0.989/0.934	0.970/0.955	0.883/0.420	0.959/0.856

both datasets. However, choosing too high a community level can strip away necessary details.

- **GraphRAG Local** offers a practical, low-overhead alternative for queries that do not require global, multi-hop reasoning yet still benefit from cross-entity linking; it maintains a small computational footprint while accurately handling many comparison-style questions.
- **Hybrid RAG** remains a strong baseline across datasets: combining dense and sparse retrieval often yields balanced Precision and Recall without the overhead of graph-based summarisation.
- **Prompt tuning** delivers consistent gains at negligible cost. Using a grounded, concise template reduces hallucinations and encourages succinct answers, directly improving Faithfulness and Context Precision in both settings.
- Easy questions saturate quickly on both datasets, with diminishing returns from complex retrieval schemes. Medium and more challenging questions highlight the advantages of GraphRAG and prompt tuning.

Chapter 6

Conclusion

This thesis has investigated how graph-based retrieval-augmented generation can improve question answering over complex document collections, with a particular focus on scenarios that resemble enterprise knowledge management. Starting from the limitations of classical RAG pipelines based solely on dense or sparse retrieval, the work explored how an explicit knowledge graph and community structure can help large language models reason across multiple documents and connect related pieces of information.

To this end, three baseline RAG systems were implemented: a sparse retriever based on lexical matching, a dense retriever using vector embeddings and a hybrid system combining both signals. On top of these baselines, a GraphRAG pipeline was integrated and adapted, including two query modes: *Local*, designed to focus on graph neighborhoods around specific entities, and *Drift*, aimed at global, sense-making queries that benefit from community-level summaries. Particular attention was devoted to the practical implementation aspects, such as document ingestion and chunking, embedding and vector storage, prompt design and the integration of RAG-specific evaluation.

The experimental evaluation relied on two datasets: a subset of WikiEval and a tailored subset of HotpotQA. WikiEval provided a collection of questions that, while non-trivial, often did not require deeply global reasoning. By contrast, the selected HotpotQA questions were explicitly multi-hop and included both bridge and comparison types. Across both datasets, the systems were evaluated using a suite of metrics from the RAGAs framework, measuring answer faithfulness, answer relevancy, context precision and context recall. The experiments also compared two prompt variants, including a grounded template aimed at reducing hallucinations and encouraging concise answers and examined the effect of different GraphRAG community levels on performance.

6.1 Main Achievements and Limitations

From the implementation and experiments, several contributions and observations emerge. First, the work shows that a graph-based approach such as GraphRAG can provide tangible benefits over classical RAG baselines in settings where questions require connecting information across multiple documents, entities, or topics. In particular, the Drift mode consistently achieved higher faithfulness and context recall on more demanding, multi-hop questions, especially in the HotpotQA subset. This indicates that leveraging community-level summaries and a global view of the corpus enables the model to retrieve and combine more of the relevant evidence needed to answer complex queries.

Second, the Local mode of GraphRAG proved to be a competitive and cost-effective alternative. While it does not expose the same degree of global, sense-making behaviour as Drift, it performed strongly on questions where the reasoning remained within a relatively contained subset of the graph but still required traversing several connections. Local therefore appears well suited for many practical queries that are not fully global but benefit from structured, cross-entity linking. Importantly, Local tends to be less computationally demanding than Drift, making it attractive in scenarios where resources or latency are constrained.

Third, the experiments confirm the intuition that hybrid retrieval remains a strong classical baseline. Combining sparse and dense retrievers yields balanced performance across relevancy and recall and it often surpasses purely sparse or purely dense approaches. Nonetheless, even a strong hybrid baseline struggled to match GraphRAG on the most complex questions, especially those requiring multi-hop reasoning over disparate parts of the corpus. Sparse retrieval in particular was disadvantaged when the key terms of the question and the evidence were lexically distant or paraphrased, a situation where semantic and graph-based methods have a clear advantage.

A further achievement of this thesis is the systematic use of prompt tuning to improve generation quality. The grounded prompt variant, which explicitly constrains the model to rely on the retrieved context and, in some settings, limits the length of the answer, consistently improved faithfulness and context precision across systems. This effect was visible both in classical RAG pipelines and in GraphRAG, confirming that even simple prompt-level interventions can meaningfully reduce hallucinations and encourage the model to behave more like an extractive or grounded QA system.

The study of community levels within GraphRAG adds another layer of insight. The results suggest that intermediate levels typically offer the best trade-off between detail and coverage. Very fine-grained (leaf-level) communities provide rich local detail but can fragment the evidence, making it harder to assemble a coherent answer without processing many summaries. Very coarse levels, on the other

hand, risk over-compressing information and diluting critical facts, especially in the more challenging HotpotQA questions. The empirical findings support the idea that choosing an appropriate community level is an important hyperparameter in GraphRAG and that a small range of intermediate levels often yields the most robust performance.

Finally, the RAGAs metrics and the qualitative error analysis helped to characterise where the systems succeed and where they fail. GraphRAG Drift tends to excel when the answer requires integrating multiple pieces of evidence spread across the corpus, while Local shines on moderately complex questions with a clear subgraph structure. Classical baselines are reasonably strong on simpler or lexically well-aligned questions but degrade more quickly as reasoning complexity increases. Frequent errors involved missed entity linking, incomplete retrieval for comparison questions and over-compression at higher community levels, all of which point to concrete directions for future refinement.

Alongside these achievements, several limitations remain. Despite the improvements brought by GraphRAG, some HotpotQA questions proved too difficult even for the Drift mode: when the reasoning chain becomes very long or when crucial evidence is only weakly represented in the graph, the system can still fail to recover all necessary information or compose it correctly. Moreover, the graph-based pipeline is computationally demanding. Building and maintaining the knowledge graph, running community detection and generating hierarchical summaries require substantial resources, which makes scaling to very large, frequently changing corpora challenging. In particular, when new data are added, the structure of the knowledge base may need to be recomputed or significantly updated, so efficient incremental update strategies remain an open problem.

At a more conceptual level, the limitations of RAG-style systems are not completely removed by GraphRAG. The model is still fundamentally constrained by the quality of retrieval and by the underlying language model's ability to perform reliable reasoning over the provided context. Recent trends in the field are moving towards agentic systems that can plan, decompose tasks and interact with tools in multiple steps, rather than relying on a single retrieval-and-answer pass. GraphRAG can be seen as a powerful retrieval component within such a broader architecture, but by itself it does not address all aspects of long-horizon reasoning, interaction, or adaptation.

These achievements and limitations suggest that, while GraphRAG represents a meaningful step forward compared to classical RAG baselines, there is substantial room for further development. The next section outlines several directions for future work, including deployment in enterprise environments, integration into AI agents, and deeper connections with graph databases and operational data sources.

6.2 Future Directions

The work carried out in this thesis is primarily experimental and research-oriented, but it is motivated by real enterprise needs. A natural direction for future development is therefore the deployment of this kind of system in a company setting. In such a context, GraphRAG could operate over internal documentation, code repositories, tickets, or logs, helping new employees navigate complex systems and supporting experienced staff in cross-domain investigations. This deployment would require robust integration with authentication, access control and monitoring, but the benefits could be substantial in terms of reduced onboarding time and improved access to institutional knowledge.

A second promising avenue is to integrate GraphRAG into the toolkit of an AI agent. Instead of being invoked as a single-step retrieval-and-answer module, GraphRAG could be used by an agent capable of planning, decomposing tasks and iteratively calling tools. In this setting, the agent might use GraphRAG for high-precision, structured retrieval when it suspects that the answer depends on complex relationships or long-range dependencies, while relying on simpler vector search or direct model reasoning for other queries. Such an agent could engage in multi-turn dialogues, ask clarification questions and refine its own search over the graph, yielding a more interactive and adaptive system for end-users.

The integration with graph databases such as Neo4j also offers interesting opportunities, particularly in domains where structured relationships are central. In sectors like insurance, finance or logistics, data is often naturally represented as a graph: policies linked to customers, claims linked to events, transactions linked to entities and so on. Combining an operational Neo4j graph with GraphRAG-style summarisation and retrieval would allow the system to ground its answers not only in unstructured documents but also in the structured, transactional backbone of the organisation. This could be useful for tasks such as exploring risk relationships, detecting patterns across cases, or answering complex questions that combine textual and relational information.

There are, moreover, several methodological extensions that could further strengthen the approach. One is the development of adaptive mechanisms for selecting the community level or even dynamically combining information across levels based on the nature of the query. Another is to expand the evaluation beyond the current datasets and metrics, incorporating larger and more diverse corpora, as well as human-in-the-loop assessments that capture user satisfaction and practical utility. A further direction is to study how the graph and its summaries can be incrementally updated as new documents arrive, enabling the system to remain current without rebuilding the entire index.

In conclusion, this thesis has demonstrated that graph-based retrieval-augmented generation is a promising path for addressing complex, cross-document questions

in realistic settings. By combining a structured representation of knowledge with modern language models and carefully designed retrieval and prompt strategies, the proposed system advances the state of the art beyond classical RAG baselines. At the same time, it opens up a rich space of future work, from real-world deployment and agent integration to deeper methodological refinement. The hope is that these results and ideas will contribute to the development of more capable, reliable and transparent AI systems for knowledge-intensive work.

Bibliography

[1] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv preprint arXiv:2005.14165. 2020. URL: https://arxiv.org/abs/2005.14165 (cit. on pp. 9, 12).

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. *Attention Is All You Need*. arXiv preprint arXiv:1706.03762. 2017. URL: https://arxiv.org/abs/1706.03762 (cit. on p. 9).

[3] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. *BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension*. arXiv preprint arXiv:1910.13461. 2019. URL: https://arxiv.org/abs/1910.13461 (cit. on p. 11).

[4] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. *Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer*. arXiv preprint arXiv:1910.10683. 2019. URL: https://arxiv.org/abs/1910.10683 (cit. on p. 11).

[5] Microsoft Research. *GraphRAG vs Baseline RAG*. 2025. URL: https://microsoft.github.io/graphrag/ (cit. on pp. 12, 21, 22).

[6] Anonymous. *Hybrid RAG Systems: Design, Comparison, and Application*. arXiv preprint arXiv:2404.09611. 2024. URL: https://arxiv.org/abs/2404.09611 (cit. on p. 14).

[7] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2020. URL: https://arxiv.org/abs/2005.11401 (cit. on pp. 14, 15).

[8] Wayne *et al.* Zhao. *A Survey on Retrieval-Augmented Generation*. arXiv preprint arXiv:2305.01569. 2024. URL: https://arxiv.org/abs/2305.01569 (cit. on pp. 15, 17).

[9] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. *Generalization through Memorization: Nearest Neighbor Language Models*. arXiv preprint arXiv:1911.00172. 2019. URL: https://arxiv.org/abs/1911.00172 (cit. on p. 16).

[10] D. Edge, A. Singh, S. Agarwal, et al. *From Local to Global: A Graph RAG Approach to Query-Focused Summarization*. arXiv preprint arXiv:2404.16130. 2025. URL: https://arxiv.org/abs/2404.16130 (cit. on pp. 21, 36, 40, 41).

[11] Vincent A. Traag, Ludo Waltman, and Nees Jan van Eck. «From Louvain to Leiden: Guaranteeing well-connected communities». In: *Scientific Reports* 9.1 (2019), p. 5233. DOI: 10.1038/s41598-019-41695-z. URL: https://www.nature.com/articles/s41598-019-41695-z (cit. on pp. 22, 38).

[12] Zhen Hu and Jiefu *et al.* Lu. *Evaluation of RAG Systems: Metrics and Benchmarks*. arXiv preprint arXiv:2310.07994. 2024. URL: https://arxiv.org/abs/2310.07994 (cit. on pp. 22, 46).

[13] Davide Ravasio, Marcin Dubiel, and Jonathan Raiman. *RAGAs: An Open-Source Framework for Holistic RAG Evaluation*. arXiv preprint arXiv:2310.12082. 2024. URL: https://arxiv.org/abs/2310.12082 (cit. on pp. 22, 44–46).

[14] Patrick Lewis et al. «Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks». In: *Advances in Neural Information Processing Systems (NeurIPS 2020)*. 2020, pp. 9459–9474. URL: https://arxiv.org/abs/2005.11401 (cit. on pp. 24, 30, 34).

[15] Harrison Chase. *LangChain*. GitHub repository. 2022. URL: https://github.com/langchain-ai/langchain (cit. on p. 25).

[16] Stephen Robertson and Hugo Zaragoza. «The Probabilistic Relevance Framework: BM25 and Beyond». In: *Foundations and Trends in Information Retrieval* 3.4 (2009), pp. 333–389. URL: https://doi.org/10.1561/1500000019 (cit. on pp. 26, 27).

[17] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Buttcher. «Reciprocal Rank Fusion Outperforms Condorcet and Individual Rank Learning Methods». In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2009, pp. 758–759. URL: https://doi.org/10.1145/1571941.1572114 (cit. on pp. 26, 28, 36).

[18] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. «Dense Passage Retrieval for Open-Domain Question Answering». In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2020, pp. 6769–6781. URL: https://arxiv.org/abs/2004.04906 (cit. on pp. 27, 34).

[19] Priyanka Mandikal and Raymond Mooney. «Sparse Meets Dense: A Hybrid Approach to Enhance Scientific Document Retrieval». In: *Proceedings of the 4th Workshop on Scientific Document Understanding (SDU) at AAAI 2024*. 2024. URL: https://arxiv.org/abs/2401.02419 (cit. on pp. 28, 36).

[20] Exploding Gradients. *WikiEval: An Evaluation Benchmark for Retrieval-Augmented Generation*. Available on Hugging Face Datasets. 2024. URL: https://huggingface.co/datasets/explodinggradients/WikiEval (cit. on pp. 44, 46).

[21] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. «HotpotQA: A Dataset for Diverse, Explainable Multi-hop Question Answering». In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 2018, pp. 2369–2380. URL: https://arxiv.org/abs/1809.09600 (cit. on p. 46).