FernUniversität in Hagen
Faculty of Mathematics and Computer Science

AIG
Artificial Intelligence Group

Enhancing Retrieval-Augmented Generation with Knowledge Graphs for Domain-Specific Technical Support

Master's Thesis

in partial fulfillment of the requirements for
the degree of Master of Science (M.Sc.)
in Data Science

submitted by
Nasiba Tuychieva

First examiner:	Prof. Dr. Matthias Thimm Artificial Intelligence Group
Advisor:	Prof. Dr. Matthias Thimm Artificial Intelligence Group

Statement

I declare that I have written the master´s thesis independently and without unauthorized use of third parties. I have only used the indicated resources and I have clearly marked the passages taken verbatim or in the sense of these resources as such. The assurance of independent work also applies to any drawings, sketches or graphical representations. The work has not previously been submitted in the same or similar form to the same or another examination authority and has not been published. By submitting the electronic version of the final version of the master´s thesis, I acknowledge that it will be checked by a plagiarism detection service to check for plagiarism and that it will be stored exclusively for examination purposes.

I explicitly agree to have this thesis published on the webpage of the artificial intelligence group and endorse its public availability.

Software created for this work has been made available as open source; a corresponding link to the sources is included in this work. The same applies to any research data.

Düsseldorf, 08.01.2026 .....

(Place, Date) (Signature)

Zusammenfassung

Retrieval-augmented Generation (RAG) hat sich als ein vielversprechender Ansatz für domänenspezifische KI-Anwendungen etabliert, insbesondere in Szenarien, in denen große Sprachmodelle (Large Language Models, LLMs) inhärente Einschränkungen aufweisen, etwa im Umgang mit aktuellem, domänenspezifischem Wissen oder bei der intensiven Verarbeitung strukturierter und unstrukturierter Dokumente. Durch die Erweiterung von Sprachmodellen um externe Retrieval-Mechanismen ermöglicht RAG die Einbindung relevanter Kontextinformationen zur Inferenzzeit. Die Weiterentwicklung robuster und verlässlicher RAG-Systeme stellt ein aktives Forschungsfeld dar, wobei die Integration von Wissensgraphen (KGs) einen von mehreren Ansätzen zur Einbindung strukturierter und relationaler Informationen darstellt.

Diese Arbeit untersucht die Integration von Wissensgraphen (KGs) und RAG im Kontext LLM-basierter Chatbots für technische Produktdokumentationen. Zunächst werden die theoretischen Grundlagen von RAG und KGs dargestellt, gefolgt von einer vergleichenden Betrachtung unterschiedlicher Ansätze zur Konstruktion von Wissensgraphen und deren zugrunde liegenden Methodiken. Anschließend wird analysiert, ob und in welchem Ausmaß KG-erweiterte RAG-Ansätze die Antwortqualität über verschiedene Fragetypen hinweg verbessern können. Zu diesem Zweck wird ein modulares experimentelles Setup implementiert, das die Eigenschaften und das Verhalten verschiedener Systemvarianten untersucht, darunter klassische RAG-Pipelines sowie KG-erweiterte RAG-Ansätze.

Abstract

Retrieval-augmented generation (RAG) has emerged as a promising approach for domain-specific AI applications, particularly in scenarios where large language models (LLMs) face inherent limitations, such as the need for up-to-date, domain-specific knowledge or intensive interaction with structured and unstructured documents. By augmenting language models with external retrieval mechanisms, RAG enables the incorporation of relevant contextual information at inference time. Improving the robustness and reliability of RAG-based systems is an active research topic, and the integration of knowledge graphs (KGs) represents one approach to incorporating structured and relational context into retrieval.

This thesis investigates the integration of knowledge graphs (KGs) and RAG in the context of LLM-based chatbots for technical product documentation. It first provides a theoretical foundation of RAG and KGs, followed by a comparative discussion of different KG construction approaches, highlighting their underlying methodologies. Subsequently, it is examined whether and to what extent KG-enhanced RAG can improve answer quality across different types of queries. To this end, a modular experimental setup is implemented to examine the behavior and characteristics of different system variants, including standard RAG pipelines and KG-enhanced RAG approaches.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Recent advances in Artificial Intelligence (AI), particularly in Large language models (LLMs), have led to significant changes in how complex problems are approached in both research and industry. These developments respond to the increasing need for generalized models that can effectively handle a wide range of language-related tasks, including translation, summarization, information retrieval (IR), and conversational interactions. Such requirements have been met by recent breakthroughs in language modeling, which have been driven primarily by the introduction of transformer-based architectures [13], increased computational capabilities, and the availability of large-scale training data. These advances have enabled the development of LLMs that demonstrate strong performance across a variety of benchmark tasks and, in some cases, approach human-level performance [97, 3]. As a result, LLMs have emerged as state-of-the-art AI systems capable of generating coherent natural language [6] while generalizing across multiple tasks without task-specific training [80, 12]. LLMs are trained on large-scale textual data collected from diverse sources and are therefore able to capture a substantial amount of factual knowledge about the world, ranging from well-known facts to more specialized, domain-specific information. This knowledge is implicitly encoded in the model parameters rather than stored in an explicit or structured form [78]. Given the scale of contemporary pre-training datasets and model architectures, it is reasonable to expect that LLMs can acquire a vast amount of information from textual data, including knowledge that is infrequently mentioned in the training corpus [46]. Consequently, LLMs are inherently limited by the temporal scope of their training data and therefore cannot reliably answer questions about events that occurred after the completion of their training phase [27]. Given the scale of contemporary pre-training datasets and the size of modern LLMs , it is reasonable to expect that these models can acquire a vast amount of information from large-scale textual data collected from diverse sources. However, the distribution of knowledge within such datasets is inherently uneven, as not all information occurs with the same frequency across the underlying sources. In particular, a substantial portion of factual knowledge lies in the long-tail of the knowledge distribution, referring to specialized or infrequently mentioned information that is underrepresented in the training data [46]. In practice, a substantial portion of this long-tail knowledge consists of highly domain-specific and industry-relevant information that, despite its low frequency in training data, is critical for specialized applications. This highlights the need for structured frameworks to manage expert-level information more reliably [51].

In companies that develop or manufacture technical products, customer support teams are often confronted with the challenge of resolving complex technical issues based on fragmented and heterogeneous documentation. When customers request assistance, support staff must rapidly access and interpret both structured informa-

tion—such as configuration data or component lists—and unstructured content like manuals or installation guides. The scattered nature of this domain-specific knowledge across formats and systems complicates efficient problem resolution, emphasizing the need for intelligent tools that can retrieve and contextualize relevant technical information on demand.

Moreover, a substantial portion of enterprise data is proprietary, confidential, or otherwise not publicly accessible and therefore not included in the training corpora of LLMs. As a consequence, LLMs lack exposure to organization-specific documentation, internal processes, and highly specialized technical knowledge. This limitation significantly constrains their ability to provide accurate and reliable answers in enterprise and domain-specific settings, where relevant information is often unique, rarely mentioned in public sources, and subject to frequent updates. Relying solely on the parametric knowledge of LLMs is thus unlikely to yield high-quality responses for technical customer support scenarios without access to external, up-to-date enterprise knowledge.

Retrieval-Augmented Generation (RAG, as introduced by Lewis et al. [56], combines the generative capabilities of LLMs with external, non-parametric knowledge sources, such as document collections or databases, in order to improve performance on knowledge-intensive tasks. By conditioning the generation process on retrieved evidence, RAG enables language models to access information that is not contained within their parametric memory.

Beyond improving factual coverage, RAG also offers a cost-effective and flexible alternative to continual pretraining or fine-tuning of LLMs. As demonstrated by Lewis et al. [56], the framework allows for the dynamic incorporation of up-to-date information through traditional retrieval mechanisms or auxiliary language models, without requiring direct integration of new data into the model parameters. Since the retrieved evidence is drawn from real-world, human-authored sources, RAG further enhances the reliability and factual consistency of generated outputs.

1.2 Problem Statement

Traditional RAG systems frequently encounter limitations when queries require multi-hop reasoning or when crucial context is dispersed across multiple documents and data formats [51]. Conventional RAG systems often fall short when user queries demand multi-step reasoning or when essential contextual information is spread across heterogeneous sources. To overcome these limitations, recent approaches propose integrating knowledge graphs (KGs) into the RAG pipeline. By organizing information into semantically linked entities and relations, KGs facilitate more targeted retrieval and structured reasoning—particularly useful for capturing and navigating long-tail, domain-specific knowledge [70] [98].

While RAG and KGs offer promising techniques to enhance the capabilities of LLMs, their practical integration remains a challenge—particularly in specialized domains. RAG systems based on unstructured retrieval often fail to capture deeper

semantic relations or resolve ambiguity in complex queries. Conversely, KGs enable structured reasoning but require careful design and integration into dynamic, language-based systems.

Despite initial research efforts, there is limited empirical understanding of how combining RAG and KGs affects the factual accuracy, contextual depth, and usability of responses in technical applications. Moreover, there is a clear need for systematic comparisons between standalone RAG architectures and hybrid approaches to assess their relative effectiveness and practical feasibility. This thesis aims to address this gap by investigating the strengths and limitations of both paradigms—individually and in combination—in the context of domain-specific question answering.

1.3 Research Questions and Contributions

Based on the challenges outlined in the previous section, this thesis formulates a set of research questions to explore whether and under which conditions hybrid approaches combining RAG and KGs can support domain-specific technical question answering. Rather than assuming inherent benefits, the thesis adopts an exploratory perspective on the integration of structured knowledge into RAG-based systems.

The first line of inquiry examines the construction and quality of automatically generated KGs. In particular, it investigates how KGs can be derived from unstructured technical documentation and assesses the reliability and accuracy of the resulting entities and relations.

A second focus concerns the potential impact of KGs on the performance of RAG-based systems. This includes analyzing whether and to what extent the incorporation of structured knowledge influences answer quality, as well as identifying possible effects on factual accuracy, relevance, and reasoning behavior in chatbot responses.

Third, the thesis explores the usefulness of graph-augmented retrieval across different types of information needs. The objective is to determine which classes of queries—such as entity disambiguation or multi-hop—may benefit from the inclusion of structured semantic relations, and which may not.

Finally, the thesis addresses design and implementation considerations for KG-enhanced RAG systems. This includes examining how data sources should be prepared to support effective graph construction, how such hybrid architectures can be instantiated in technical or industrial environments, and how their performance and usability compare to a standard RAG baseline.

1.4 Structure of the Thesis

The remainder of this thesis is structured as follows. Section 2 provides the theoretical background, introducing LLMs, RAG, and KGs, and discusses relevant related work. Section 3 describes the methodological framework and system design, including the dataset, data preparation steps, the baseline RAG pipeline, the construction

and refinement of the KG, and the architecture of the hybrid KG-enhanced RAG system. Section 4 presents the feasibility study, outlining the design of the question set, the gold-standard answers, the assessment method using an LLM-as-a-Judge, and the findings regarding the usefulness and limitations of the developed system variants. Finally, Section 5 concludes the thesis by providing an overall summary and concluding remarks.

2 Theoretical Background

This chapter introduces the theoretical concepts that underpin the approaches investigated in this thesis. In particular, it discusses the machine learning foundations and transformer architectures underlying LLMs, as well as their role in knowledge-intensive tasks. These concepts provide the basis for understanding retrieval mechanisms and RAG, including the integration of structured knowledge sources such as KGs.

2.1 Foundations of Large Language Models

RAG builds upon the capabilities of LLMs, which serve as the fundamental generative component responsible for interpreting user queries and producing natural-language responses. Understanding the principles and foundations of LLMs is therefore essential before examining RAG and its integration with external knowledge sources. LLMs are commonly defined as transformer-based language models with a very large number of parameters, often reaching hundreds of billions or more, which are trained on massive collections of textual data [87]. Representative instances of such models include GPT-3 [12], PaLM [15], Galactica [93], and LLaMA [94]. Due to their scale and training regime, LLMs demonstrate advanced capabilities in natural language understanding and generation, enabling them to address a wide range of complex language-based tasks.

From a formal perspective, an LLM can be described as a neural network that processes a given textual context and produces a probability distribution over possible subsequent tokens. The fundamental principle of language modeling is that a system capable of assigning probabilities to sequences of text can also be used for text generation by iteratively sampling from the predicted distribution of the next tokens [45].

From an architectural perspective, contemporary LLMs are predominantly based on the Transformer architecture [96], in which multiple multi-head self-attention layers are stacked to form deep neural networks. Despite their substantially increased scale, existing LLMs largely follow architectural principles and pre-training objectives—most notably autoregressive language modeling—that are conceptually similar to those employed in smaller-scale language models [105].

Tokenization. Tokenization [99] constitutes a fundamental preprocessing step in the training of LLMs, in which raw textual input is segmented into discrete, non-overlapping units referred to as tokens. Depending on the chosen tokenization strategy, tokens may correspond to characters, subword units [53], symbols [85], or complete words. Modern LLMs predominantly rely on subword-based tokenization schemes in order to achieve a balance between vocabulary size and representational expressiveness. Commonly used approaches include WordPiece tokenization [90], Byte Pair Encoding (BPE) [85, 30], and the Unigram Language Model tokenizer [53].

Activation Functions. Activation functions play a central role in determining the expressive capacity of neural networks by introducing non-linear transformations into the model [63]. In the context of large language models (LLMs), certain activation functions have proven particularly effective and are therefore briefly discussed below.

Rectified Linear Unit (ReLU) is one of the most widely used activation functions in deep neural networks and is defined as

$$\mathrm{ReLU}(x) = \max(0, x)$$

where the operation is applied element-wise to the input. Despite its simplicity, ReLU enables efficient optimization and has historically been adopted in a wide range of neural architectures [63].

Gaussian Error Linear Unit (GeLU) extends the ReLU activation by incorporating a smooth probabilistic gating behavior that resembles stochastic regularization effects observed in methods such as dropout [91] and zoneout [52]. GeLU has been shown to improve performance in transformer-based language models and is therefore commonly employed in contemporary LLM architectures [38, 63].

Gated Linear Units (GLU) and their variants introduce a gating mechanism that modulates the output of a linear transformation through an element-wise product with a sigmoid-activated projection [18]. Formally, a GLU layer is defined as

$$\mathrm{GLU}(x, W, V, b, c) = (xW + b) \otimes \sigma(xV + c),$$

where x denotes the input to the layer, W and V are weight matrices, b and c are bias terms, $\sigma(\cdot)$ is the sigmoid function, and $\otimes$ denotes element-wise multiplication [63]. Several GLU variants have been proposed and adopted in LLMs to further improve representational capacity [88]. Common variants include

$$\mathrm{ReGLU}(x, W, V, b, c) = \max(0, xW + b) \otimes (xV + c),$$

$$\mathrm{GEGLU}(x, W, V, b, c) = \mathrm{GELU}(xW + b) \otimes (xV + c),$$

$$\mathrm{SwiGLU}(x, W, V, b, c, \beta) = \mathrm{Swish}_\beta(xW + b) \otimes (xV + c),$$

where each variant combines a different non-linear activation with the gating mechanism. These formulations are commonly employed in modern transformer-based LLMs due to their favorable optimization behavior and empirical performance characteristics [63].

Encoding Positions. Transformer-based models process input tokens in parallel and, by design, do not inherently encode information about the sequential order of tokens. In particular, the self-attention mechanism is permutation-invariant and therefore unable to capture positional relationships on its own. To address this limitation, positional encodings were introduced as part of the Transformer architecture [96]. These encodings inject information about token positions by combining a positional embedding with the corresponding token embedding.

Several variants of positional encoding have been proposed, including *absolute*, *relative*, and *learned* positional representations. In the context of large language models, relative positional encoding schemes have gained particular importance. Two widely adopted approaches are *ALiBi* and *Rotary Positional Embeddings (RoPE)*. *ALiBi* [79] incorporates positional information by adding a position-dependent bias to the attention scores, where the bias increases with the distance between token positions. This mechanism encourages the model to place greater emphasis on more recent tokens during attention computation. *RoPE* [92] encodes positional information by applying a rotation to the query and key representations, with the rotation angle determined by the absolute position of each token. This results in an implicit relative positional encoding scheme in which the interaction between tokens depends on their relative positional distance.

2.2 The Architecture of Large Language Models

The architecture of LLMs has a substantial influence on their performance, computational efficiency, and scalability. In general, LLM architectures can be characterized by a set of core components, most notably the encoder and the decoder. In encoder–decoder and encoder-only architectures, the encoder constitutes a fundamental building block, as it processes input sequences and projects them into a high-dimensional representation space, thereby capturing contextual dependencies within the data [59].

Architecturally, an encoder is typically realized as a stack of identical layers. Each layer consists of two principal subcomponents: a multi-head self-attention mechanism and a position-wise fully connected feed-forward network, which together enable the modeling of complex contextual relationships across the input sequence [96]. In contrast, the decoder is responsible for generating output sequences, either conditioned on representations produced by an encoder or directly on previously generated tokens [59]. In LLMs such as GPT-3 [12] and its successors, the decoder operates according to an autoregressive modeling paradigm, in which each output token is predicted sequentially based on the tokens generated in previous steps.

A defining characteristic of decoder architectures in LLMs is causality, which constrains the model to attend only to past tokens when predicting the current token. This constraint prevents access to future information and is enforced through masked self-attention mechanisms within the decoder layers of the Transformer architecture [96]. Depending on their architectural components and connectivity

patterns, LLMs can be broadly classified into three main categories: encoder-only, decoder-only, and encoder–decoder architectures. Despite these structural differences, all three categories can be understood within the broader sequence-to-sequence (seq2seq) modeling paradigm, which encompasses models that map input sequences to output sequences [59].

2.2.1 Encoder–Decoder Architecture

The original Transformer architecture introduced by Vaswani et al. [96] belongs to the class of encoder–decoder models. In this architecture, the encoder maps an input sequence into a set of continuous representations that capture both semantic and syntactic characteristics of the input.

The decoder subsequently generates an output sequence based on these encoded representations. Output tokens are produced in an autoregressive manner, where each token is predicted by conditioning on the previously generated tokens as well as the encoder output. The integration of cross-attention mechanisms enables the decoder to selectively attend to relevant parts of the encoded input during generation. This separation of encoding and decoding processes allows the encoder–decoder architecture to flexibly address a wide range of language processing tasks by decoupling input understanding from output generation [59]. The encoder–decoder Transformer model can be formalized as follows.

Definition 1 (Encoder–Decoder Model)**.** Let $(x_1, \ldots, x_n)$ denote an input sequence of tokens and $(y_1, \ldots, y_m)$ an output sequence. An encoder–decoder Transformer defines a conditional sequence model that learns the probability distribution

$$p(y_1, \ldots, y_m \mid x_1, \ldots, x_n).$$

The encoder maps the input sequence to a sequence of contextual representations

$$(h_1^{(e)}, \ldots, h_n^{(e)}) = \mathrm{Encoder}(x_1, \ldots, x_n),$$

where each $h_i^{(e)} \in \mathbb{R}^d$ captures contextual information about the input token x_i. The decoder generates the output sequence autoregressively according to

$$p(y_1, \ldots, y_m \mid x_1, \ldots, x_n) = \prod_{t=1}^{m} p(y_t \mid y_1, \ldots, y_{t-1}, h_1^{(e)}, \ldots, h_n^{(e)}),$$

where each conditional probability is computed by a decoder function that incorporates masked self-attention over previously generated tokens and cross-attention over the encoder representations.

2.2.2 Causal and Prefix Decoder

A causal decoder generates output tokens autoregressively, ensuring that each token is predicted solely based on previously generated tokens by enforcing a unidirectional attention mask. This causality constraint prevents the model from accessing

future information and preserves the autoregressive property of transformer-based decoders. The prefix decoder extends this mechanism by allowing bidirectional attention over a fixed input prefix while maintaining unidirectional attention for generated tokens. This modified masking scheme enables the model to condition generation on both the input prefix and the previously generated output tokens. Such architectures are particularly useful for tasks requiring controlled or prefix-conditioned sequence generation [59].

2.2.3 Transformer Architecture

The Transformer architecture has become the prevailing foundation for LLMs due to its ability to effectively model complex linguistic structures and capture long-range dependencies [96]. This architectural design enables the training of models at unprecedented scales, reaching billions or even trillions of parameters [12]. A Transformer model is typically composed of a stack of identical layers, each consisting of a multi-head self-attention sublayer and a position-wise fully connected feed-forward network [96].

Given an input sequence of tokens $(x_1, \ldots, x_n)$, the Transformer architecture maps this sequence to a set of contextualized vector representations by means of stacked attention-based layers. Each input token x_i is first mapped to a continuous vector representation and combined with positional information, yielding

$$z_i^{(0)} = e(x_i) + p_i,$$

where $e(x_i) \in \mathbb{R}^d$ denotes the token embedding and $p_i \in \mathbb{R}^d$ represents the positional encoding associated with token position i. For each Transformer layer l, self-attention is computed by projecting the input representations of the previous layer into query, key, and value spaces:

$$Q = Z^{(l-1)}W_Q, \quad K = Z^{(l-1)}W_K, \quad V = Z^{(l-1)}W_V,$$

where $Z^{(l-1)} = (z_1^{(l-1)}, \ldots, z_n^{(l-1)})$. The attention operation aggregates contextual information according to

$$\mathrm{Att}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V,$$

with d_k denoting the dimensionality of the key vectors. To enable the model to capture multiple types of dependencies in parallel, multi-head attention is employed by computing several attention heads independently and concatenating their outputs:

$$\mathrm{MHA}(Z^{(l-1)}) = \mathrm{Concat}(\mathrm{head}_1, \ldots, \mathrm{head}_h)W_O.$$

Following the attention sublayer, a position-wise feedforward network applies a non-linear transformation to each token representation:

$$\mathrm{FFN}(z) = \sigma(zW_1 + b_1)W_2 + b_2,$$

where $\sigma(\cdot)$ denotes a non-linear activation function. A complete Transformer layer is then defined as

$$\tilde{Z}^{(l)} = \mathrm{MHA}(Z^{(l-1)}) + Z^{(l-1)}, \qquad Z^{(l)} = \mathrm{FFN}(\tilde{Z}^{(l)}) + \tilde{Z}^{(l)},$$

with residual connections and layer normalization applied after each sublayer to facilitate stable training. Stacking multiple such layers results in progressively more abstract and context-aware representations of the input sequence, forming the foundation for LLMs and their downstream applications.

2.2.4 Attention in Large Language Models

Attention mechanisms enable neural networks to assign different weights to input tokens based on their relevance, thereby allowing the model to focus on informative parts of an input sequence. In Transformer-based architectures, attention can be described as a function that maps a set of queries together with corresponding keys and values to contextualized output representations [96]. The output is computed as a weighted combination of value vectors, where the weights are determined by a compatibility function measuring the similarity between queries and keys. This mechanism constitutes a central component of LLMs.

Self-Attention. Self-attention computes attention weights using queries, keys, and values derived from the same input sequence, either within an encoder or a decoder block. This allows each token to attend to other tokens in the sequence and enables the modeling of long-range dependencies [96].

Cross-Attention. Cross-attention is primarily employed in encoder–decoder architectures. In this setting, queries are derived from decoder states, while keys and values originate from encoder outputs. This mechanism allows the decoder to condition its predictions on the encoded input sequence and is commonly used in seq2seq models.

Sparse Attention. A key limitation of standard self-attention is its quadratic time and memory complexity with respect to sequence length. To mitigate this issue, sparse attention mechanisms restrict attention computation to predefined patterns such as local or sliding windows. By limiting the number of attended positions, sparse attention reduces computational complexity while preserving relevant contextual information [14].

Flash Attention. On modern GPU architectures, memory access rather than arithmetic computation often constitutes the primary bottleneck in attention mechanisms. Flash Attention addresses this limitation by restructuring the attention computation to minimize memory reads and writes between high-bandwidth

memory and on-chip SRAM. This is achieved through input tiling and kernel fusion, resulting in substantial improvements in memory efficiency and runtime performance [17].

A simplified formulation interprets attention as a weighted aggregation of contextual representations. Given an input sequence represented by vectors $(x_1, \ldots, x_n)$, self-attention first computes *queries*, *keys*, and *values* via linear projections:

$$q_i = W_Q x_i, \qquad k_i = W_K x_i, \qquad v_i = W_V x_i.$$

In a causal setting, the attention output at position i is computed as a weighted sum over admissible previous value vectors,

$$a_i = \sum_{j \leq i} \alpha_{ij} v_j,$$

where α_{ij} denotes the attention weight assigned to position j when computing the representation at position i [96]. The weights are derived from similarity scores between the current query q_i and candidate keys k_j. In its simplest form, similarity is computed using the scaled dot product,

$$\text{score}(q_i, k_j) = \frac{q_i^\top k_j}{\sqrt{d_k}}.$$

To obtain normalized attention weights, the scores are passed through a softmax over all admissible positions,

$$\alpha_{ij} = \frac{\exp(\text{score}(q_i, k_j))}{\sum_{k \leq i} \exp(\text{score}(q_i, k_k))}, \qquad \forall j \leq i.$$

As a result, the attention output a_i represents a context-dependent embedding that aggregates information from previous tokens, weighted according to their relevance to the current token [96].

2.3 Retrieval-Augmented Generation

This section presents the fundamental concepts and core components of RAG. It provides an overview of the underlying architecture, the main processing stages, and common design choices that are relevant for understanding subsequent RAG-based methods. The discussed foundations serve as a reference for the retrieval and KG extensions analyzed later in this work.

2.3.1 Foundations

As motivated earlier, LLMs exhibit fundamental limitations when applied to enterprise-specific and domain-dependent scenarios, particularly due to their reliance on parametric knowledge acquired during pretraining. In such settings,

relevant information is often contained in private or proprietary data sources, such as technical documentation, contracts, or internal reports, which are typically not accessible during model training. As a consequence, LLM-generated responses may be unreliable and prone to hallucinations, that is, content that is inconsistent with real-world facts or the provided input [60, 43].

Retrieval-Augmented Generation (RAG) has been proposed as a principled framework to address these limitations by explicitly integrating external knowledge into the generation process [56]. Although Lewis et al. [56] build upon earlier work on incorporating external knowledge sources into neural language models [35, 47, 77], they introduced the term *Retrieval-Augmented Generation* and formalized a unified framework that combines pretrained parametric memory, represented by the language model, with non-parametric memory in the form of an external knowledge store. This hybrid approach enables more effective handling of knowledge-intensive tasks by grounding generation in retrieved evidence [49] and by decoupling knowledge storage from language generation.

Parametric memory refers to knowledge implicitly embedded in the model's weights during training, enabling generation based on internalized patterns. In contrast, non-parametric memory encompasses external sources—such as databases or domain-specific documents—that are not encoded in the model itself and can therefore be flexibly updated or extended. This distinction allows RAG-based systems to incorporate information that was not available during the model's training phase [49].

In addition to RAG, fine-tuning provides another mechanism for integrating proprietary or domain-specific knowledge into LLMs. Unlike RAG, which enriches the prompt with retrieved external data at inference time, fine-tuning embeds new information directly into the model's parameters, enabling concise and domain-adapted outputs. As shown in [8], fine-tuning "offers a precise, succinct output that is attuned to brevity" and is "highly effective" in enabling models to acquire new domain capabilities. However, the authors also emphasize that "the initial cost is high due to the extensive work required to fine-tune the model on new data," whereas RAG benefits from the "low initial cost of creating embeddings." The suitability of fine-tuning therefore depends on "the specific application, the nature and size of the dataset, and the resources available for model development." In practice, combining both approaches can be advantageous, as fine-tuning encodes stable domain knowledge while RAG provides flexibility, recency, and traceability of external information.

2.3.2 Retrieval-Augmented Generation Pipeline

Using a RAG architecture gives rise to a structured processing *RAG pipeline* [49], as illustrated in Figure 1. The pipeline starts with a user query and ends with a generated response, and consists of three main components: retrieval, augmentation, and generation [56, 104, 49]. During the retrieval phase, semantically relevant

documents or text segments are identified from a knowledge base—often managed in a vector database—by measuring their similarity to the input query. In the subsequent generation step, an LLM utilizes the retrieved information as contextual input to produce coherent and knowledge-grounded responses. From a conceptual perspective, RAG combines information retrieval techniques with neural seq2seq generation, thereby allowing generated outputs to be directly linked to verifiable source documents. As a result, RAG not only improves factual correctness but also enhances transparency and controllability, which are critical requirements in organizational and high-stakes application domains.

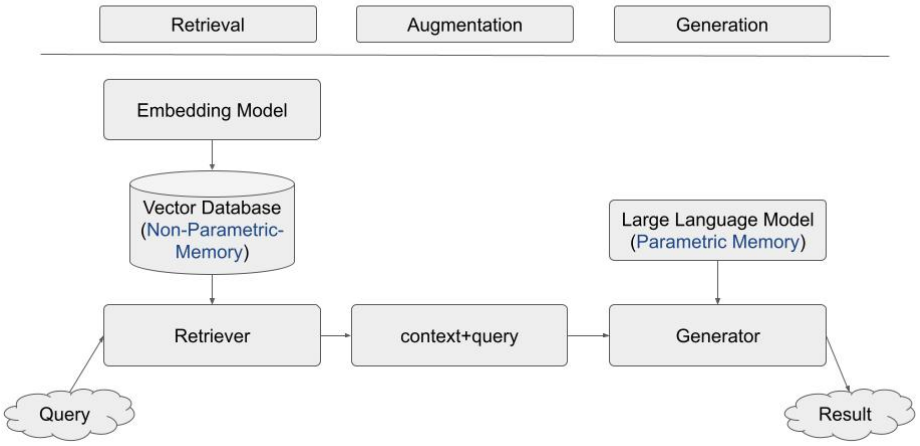


Figure 1: RAG Architecture. Adapted from [49]

The approach proposed by Lewis et al. integrates a pre-trained neural retriever—composed of a query encoder and a dense document index—with a pre-trained seq2seq generator. For an input query x, the retriever uses Maximum Inner Product Search (MIPS) to identify the top-K relevant documents z_i. The final prediction y is obtained by marginalizing over the generator's outputs conditioned on each retrieved document, treating the retrieved evidence as a latent variable. In this formulation, the Transformer-based generator is equipped with access to a dense vector index of Wikipedia through the neural retriever, and the full system is trained end-to-end as a probabilistic model in which retrieved documents influence the generation process, allowing the model to ground its outputs in external evidence rather than relying solely on parametric memory [56].

Naive RAG Pipeline. RAG pipelines can be broadly divided into three categories: Naive RAG, Advanced RAG, and Modular RAG [32]. The naive RAG pipeline refers to a *plain-vanilla* RAG architecture and represents one of the earliest and most widely

adopted variants of retrieval-augmented language modeling, gaining renewed attention with the recent widespread adoption of LLM-based assistants. This approach follows a conventional workflow consisting of document indexing, retrieval, and subsequent text generation [32]. Owing to its two-stage *inference* structure, the naive RAG framework is also commonly referred to as a *retrieve–read* paradigm [58]. Accordingly, Figure 1, which was introduced at the beginning of this section, can be interpreted as an illustration of a naive RAG pipeline as well. Advanced RAG builds on this structure by introducing optimization strategies both before and after retrieval. Pre-retrieval methods primarily focus on improving the indexing structure or refining the user query. Post-retrieval techniques include reranking retrieved chunks or compressing the context, while maintaining a largely chain-like workflow similar to Naive RAG. Modular RAG extends the paradigm further by introducing more flexible and specialized components. For example, a dedicated Search module can dynamically adapt to different scenarios by enabling direct access to search engines, databases, or KGs through LLM-generated queries or code.

2.3.3 Retrieval-Augmented Generation Models

Lewis et al. [56] formalize RAG as a probabilistic framework that integrates a neural retriever with a seq2seq generator. The retrieved document is modeled as a latent variable that is marginalized during generation. The authors propose two variants, *RAG-Sequence* and *RAG-Token*, which differ in how retrieved evidence influences the generation process and how uncertainty over relevant documents is modeled.

RAG-Sequence Model In the *RAG-Sequence* model, the generation of the output sequence y conditioned on an input sequence x is modeled by introducing a latent document variable z, which represents a document retrieved from an external knowledge source. The retrieved document is assumed to provide all relevant information required for generating the entire output sequence. As stated by the authors, the model "treats the retrieved document as a single latent variable that is marginalized to get the seq2seq probability" [56]. Formally, the conditional probability is approximated as

$$p_{\text{RAG-Sequence}}(y \mid x) \approx \sum_{z \in \text{top-}k(p_\eta(\cdot|x))} p_\eta(z \mid x)\, p_\theta(y \mid x, z).$$

In this formulation, $p_\eta(z \mid x)$ denotes the retrieval distribution, which assigns a relevance-based probability to each document z given the input x, while $p_\theta(y \mid x, z)$ represents the probability of generating the complete output sequence conditioned on both the input and the retrieved document. The summation over z corresponds to marginalization over the latent document variable. Since exact marginalization over the full document collection is computationally infeasible, it is approximated by restricting the sum to the top-k documents with the highest retrieval scores. By

conditioning the entire sequence on a single document, this formulation enforces a consistent knowledge source throughout the generation process.

RAG-Token Model In contrast to *RAG-Sequence*, the *RAG-Token* model relaxes the assumption that a single document is sufficient to inform the entire output. Instead, the model allows different retrieved documents to influence different parts of the generated sequence. As described by the authors, the model "can draw a different latent document for each target token and marginalize accordingly" [56]. This is formalized as

$$p_{\text{RAG-Token}}(y \mid x) \approx \prod_{i=1}^{m} \sum_{z \in \text{top-}k(p_\eta(\cdot|x))} p_\eta(z \mid x)\, p_\theta(y_i \mid x, z, y_{1:i-1}).$$

Here, the probability of the output sequence is factorized autoregressively over token positions i. At each decoding step, the model conditions on the previously generated tokens $y_{1:i-1}$ and marginalizes over the latent document variable z. The term $p_\theta(y_i \mid x, z, y_{1:i-1})$ denotes the probability of generating the next token given the input, the retrieved document, and the decoding history. The product over i reflects the standard autoregressive decomposition of sequence models, while the repeated marginalization over z allows different documents to contribute to different tokens. This token-level formulation enables a more fine-grained integration of external knowledge, particularly for long or complex outputs, albeit at the cost of increased computational complexity.

2.3.4 Retrieval

The first stage in the RAG pipeline is the retrieval stage, which integrates search and ranking mechanisms and is responsible for identifying and prioritizing documents from a given corpus to support high-quality text generation. It applies search algorithms to traverse indexed data structures and retrieve documents relevant to a user query. Subsequently, the retrieved documents are ranked according to their estimated relevance, ensuring that the most informative and contextually appropriate content is provided to the generation model [40].

Let $D = \{d_1, d_2, \ldots, d_M\}$ denote a collection of evidence documents representing a retrieval corpus. Given a query q, an information retrieval (IR) model selects a subset of relevant passages $Z \subset D$, one or more of which are expected to contain the correct answer to q [1]. Depending on the employed similarity functions, retrieval approaches can be broadly categorized into sparse retrieval, dense retrieval, and related variants. In both sparse and dense paradigms, the retrieval pipeline typically consists of two main stages. First, each document or data item is transformed into an appropriate representation, such as a term-based or vector-based encoding. Second, an index is constructed over these representations to enable efficient search within the underlying data collection [104].

Sparse Retriever. Sparse retrieval methods are widely applied in document retrieval scenarios, where documents constitute the searchable items. These approaches rely on lexical term-matching mechanisms, including TF–IDF [81], query likelihood models [54], and the BM25 ranking function [82]. Such methods analyze term statistics within textual data and organize documents using inverted index structures to support efficient retrieval. Among these techniques, BM25 is commonly regarded as a strong baseline for large-scale search applications, as it combines inverse document frequency weighting with term occurrence information and normalization factors [104].

Let $\mathcal{D} = \{d_1, \ldots, d_N\}$ denote a collection of documents and let $q = (t_1, \ldots, t_m)$ be a query consisting of m terms drawn from a vocabulary $\mathcal{V}$. In sparse retrieval models, both documents and queries are represented as high-dimensional sparse vectors in $\mathbb{R}^{|\mathcal{V}|}$, where each dimension corresponds to a term in the vocabulary. For a document $d \in \mathcal{D}$, the sparse representation is defined by a weighting function $w(t, d)$ that reflects the importance of term t in d. Common weighting schemes include term frequency–inverse document frequency (TF–IDF) and probabilistic models such as BM25. The relevance score between a query q and a document d is then computed as

$$\text{score}(q, d) = \sum_{t \in q} w(t, q) \cdot w(t, d),$$

where $w(t, q)$ denotes the weight of term t in the query.

To enable efficient retrieval, an inverted index is constructed that maps each term $t \in \mathcal{V}$ to a posting list containing all documents in which the term occurs. Given a query q, candidate documents are retrieved by aggregating the posting lists of the query terms and are subsequently ranked according to their relevance scores.

Dense Retriever. In contrast to sparse retrieval approaches, dense retrieval methods represent both queries and documents using continuous, low-dimensional embedding vectors learned by neural encoders. To support efficient similarity search over large document collections, dense retrieval systems typically construct an Approximate Nearest Neighbor (ANN) index over the embedded representations [104]. This paradigm is commonly referred to as Dense Passage Retrieval (DPR) [47]. Similarity between dense representations is typically computed using metrics such as cosine similarity, inner product, or Euclidean distance [104].

Formally, Dense Passage Retrieval employs two neural encoders: a passage encoder $E_P(\cdot)$ and a query encoder $E_Q(\cdot)$. The passage encoder maps each text passage p from a collection of M passages to a dense vector representation $E_P(p) \in \mathbb{R}^d$, over which an index is constructed. At inference time, the query encoder maps an input question q to a vector $E_Q(q) \in \mathbb{R}^d$. Retrieval is then performed by selecting the top-k passages whose embeddings are closest to the query embedding according to a similarity function. In DPR, similarity is commonly defined using the dot product

$$\text{sim}(q, p) = E_Q(q)^\top E_P(p),$$

and the passages with the highest similarity scores are returned as retrieval results [47].

2.3.5 Augmentation

In the augmentation stage, the retrieved context is incorporated into the input of the language model in order to condition the generation process. This is typically achieved by applying an augmentation template that specifies how the original user query is extended with additional contextual information. In particular, the retrieved context is embedded into the prompt provided to the model, often accompanied by explicit instructions guiding the model to utilize the supplied information during answer generation. A common template, for instance, states: "Use the following context: [context]. Given this context, answer the following query: [query]." The contextual component may consist of excerpts or references to the most relevant documents retrieved from the underlying vector database [49]. Based on the manner how the retriever augments the generator, existing RAG approaches can be categorized into four foundational classes [104], as summarized in Table 1.

Query-based RAG	Retrieval is performed on the user-issued query. The retrieved context is concatenated with the original query and provided as input to the generator. This is the classical form of RAG (e.g., REALM, RetDREAM).
Latent Representation-based RAG	The user query and retrieved information are fused inside the model architecture, often during encoding. Retrieval happens over latent representations rather than raw queries (e.g., RETRO, RAG-End-to-End variants).
Logit-based RAG	Retriever and generator probabilities are merged during the decoding phase. The LLM dynamically integrates retrieval results by combining token-level logits (e.g., kNN-LM, EDITSUM).
Speculative RAG	The generator runs normally, but the retriever is invoked at selected steps to correct or enhance the generation, usually to improve efficiency or reduce hallucination (e.g., REST, GPTCache, COG).

Table 1: Comparison of RAG Foundations. Adapted from [104].

2.3.6 Generator

The generator component is responsible for producing the final textual output by conditioning on the user query augmented with information retrieved from the external knowledge source. In RAG, the generator leverages both the original input query and the retrieved context to guide the generation process. Due to the strong

performance of generative models across a wide range of natural language generation tasks, the generator constitutes a central component of the overall RAG architecture. In practice, different generative models may be employed depending on the target application, with Transformer-based encoder–decoder architectures being particularly well suited for text-to-text generation tasks [104].

In the original RAG formulation, the generator is instantiated using BART, a pre-trained sequence-to-sequence Transformer model [56]. Specifically, the BART-large variant is employed, which consists of approximately 400 million parameters and follows an encoder–decoder architecture. During generation, the input query x and the retrieved document representation z are concatenated and jointly processed by the BART encoder. The parameters of the BART generator, denoted by θ, are treated as the parametric memory of the RAG model, in contrast to the non-parametric memory represented by the external document collection.

Formally, the generator defines a conditional probability distribution over output tokens. Given an input query x, a retrieved document z, and a previously generated token sequence $y_{1:i-1}$, the probability of generating the next token y_i is modeled as

$$p_\theta(y_i \mid x, z, y_{1:i-1}),$$

where θ denotes the parameters of the BART generator. The overall probability of generating an output sequence $y = (y_1, \ldots, y_m)$ is given by the autoregressive factorization

$$p_\theta(y \mid x, z) = \prod_{i=1}^{m} p_\theta(y_i \mid x, z, y_{1:i-1}).$$

The encoder component of BART maps the concatenated input sequence (x, z) to a sequence of contextual representations, while the decoder generates the output sequence token by token using masked self-attention and cross-attention over the encoder states.

2.3.7 Advanced Retrieval-Augmented Generation

Advanced RAG extends the Naive RAG paradigm by introducing targeted improvements that address its retrieval-related limitations. In particular, it enhances retrieval quality through the integration of pre-retrieval and post-retrieval strategies [32]. To improve indexing effectiveness, Advanced RAG employs techniques such as fine-grained document segmentation, sliding window indexing, and the incorporation of metadata.

Pre-retrieval approaches. The pre-retrieval stage focuses on optimizing both the indexed content and the user query. Index optimization aims to improve the quality and structure of the indexed data through increased granularity, metadata enrichment, and mixed retrieval strategies. Query optimization seeks to make user inputs more precise and retrieval-friendly, commonly via query rewriting, transformation, and expansion techniques [58, 76, 31].

We follow the rewrite–retrieve–read formulation proposed in [58] and formalize a retrieval-augmented pipeline as a three-stage process. Let $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$ denote a dataset for a knowledge-intensive task, where x_i represents an input query (e.g., a natural language question) and y_i the corresponding ground-truth output (e.g., an answer). A retrieval-augmented pipeline can be formalized as a three-stage process consisting of query rewriting, retrieval, and generation. In the query rewriting stage, the original input query x is transformed into a rewritten query $\tilde{x}$ that is intended to better express the information need required for retrieval:

$$\tilde{x} = f_{\text{rewrite}}(x),$$

where $f_{\text{rewrite}}(\cdot)$ denotes a rewriting function, typically implemented using a large language model. In the retrieval stage, the rewritten query $\tilde{x}$ is used to retrieve a set of relevant documents or context passages from an external corpus $\mathcal{C}$:

$$\mathcal{D}_{\tilde{x}} = \text{Top}_k\big(\{\text{sim}(\tilde{x}, d) \mid d \in \mathcal{C}\}\big),$$

where $\text{sim}(\cdot, \cdot)$ is a similarity function and $\text{Top}_k(\cdot)$ returns the k most relevant documents. In the final read stage, a generator produces a predicted output $\hat{y}$ conditioned on the original query x and the retrieved context $\mathcal{D}_{\tilde{x}}$:

$$\hat{y} = g(x, \mathcal{D}_{\tilde{x}}),$$

where $g(\cdot)$ denotes a generative model, such as a sequence-to-sequence transformer.

Post-retrieval approaches. The post-retrieval stage concentrates on effectively integrating retrieved context into the generation process. Common approaches include reranking retrieved chunks to prioritize the most relevant information and compressing the context to reduce redundancy. These strategies mitigate information overload and ensure that only the most salient content is provided to the language model, as implemented in frameworks such as LlamaIndex, LangChain, and Haystack [32].

Reranking constitutes an important post-retrieval step in information retrieval, where an initially retrieved set of documents is reordered to improve its relevance with respect to a given query. With the advent of pretrained language models and, more recently, large language models, reranking approaches have commonly been categorized into three main paradigms: pointwise, pairwise, and listwise methods [2]. Pointwise reranking methods independently score each query–document pair and assign a relevance score without considering other candidate documents. Pairwise reranking approaches learn to compare pairs of documents for a given query and predict which document in each pair is more relevant. Listwise reranking methods jointly consider the entire list of retrieved documents and optimize the ranking order directly with respect to a global ranking objective.

2.4 Knowledge Graphs

As discussed in the previous chapter, RAG extends the capabilities of LLMs by incorporating external knowledge sources. Despite its effectiveness, standard RAG pipelines exhibit limitations when applied to complex or domain-specific queries. In particular, reliance on unstructured text retrieval may result in incomplete or noisy context selection, challenges in multi-step reasoning, and ambiguities when multiple entities share similar lexical representations. Existing enhancements, such as reranking strategies or refined retrieval pipelines, attempt to alleviate these issues but remain fundamentally grounded in unstructured text-based retrieval.

In response to these challenges, recent research has investigated the integration of structured knowledge representations into RAG architectures. KGs constitute one such research direction, offering an explicit and relational representation of domain knowledge that can complement unstructured textual evidence. Rather than replacing conventional RAG components, KGs are explored as an additional source of contextual information whose potential benefits and limitations remain an open empirical question.

2.4.1 Conceptual Clarifications

Before examining how KGs can be combined with RAG to overcome these limitations, this chapter provides a more detailed introduction to KGs. We discuss their fundamental principles, common data models, and the theoretical foundations underlying their use in modern AI systems.

Although the term "knowledge graph" has been used in the literature since at least the early 1970s [84], its modern usage is typically traced back to Google's 2012 knowledge graph [89], as cited in [39]. The term *knowledge graph* is used with varying meanings in the literature [9, 11, 25]. Existing definitions range from concrete technical realizations to more general conceptual descriptions, and no single formulation is uniformly adopted.

In [39], various definitions of KGs are discussed. In line with [39], a KG can be understood as a graph-structured representation of real-world knowledge. It consists of nodes representing relevant entities and edges capturing semantic relationships between them. The underlying data graph follows a graph-based data model, which can take the form of a directed, edge-labeled graph, a property graph, or similar variants [39]. The encoded knowledge may originate from external sources or be derived internally from the graph itself. Hogan et al. [39] emphasize that knowledge is understood as information that is known or accepted as true in a given context.

According to Hogan et al. [39], there are two dominant data models for representing and managing KGs: the *Resource Description Framework (RDF)* and *Labeled Property Graphs (LPGs)*. A detailed comparison of these two models is provided by Donkers et al. [21]. Hogan et al. [39] distinguish four major categories of definitions used in the literature to describe the term KG. These categories reflect different historical perspectives and often conflicting interpretations of what constitutes a KG.

The first category defines a KG in its most basic form as a graph in which nodes represent entities and edges encode relations between them. This view typically assumes a directed, edge-labeled graph or, equivalently, a set of binary relations or triples. A second common definition describes a knowledge graph as a graph-structured knowledge base. Early academic use of this formulation can be found, for example, in [67] and [86]. However, this approach shifts the definitional problem to the term knowledge base, a concept rooted in the expert systems research of the 1970s and later associated with ontologies and other logical formalisms. A third category proposes additional technical criteria that a KG should fulfill. Paulheim [72], for example, characterizes KGs as descriptions of real-world entities and their relationships that provide a schema of classes and relations, enable the linking of arbitrary entities, and cover multiple subject areas. The fourth category avoids intensional definitions altogether and characterizes KGs by listing prominent examples such as DBpedia, Freebase, YAGO, or Google's KG. Hogan et al. [39] also distinguish definitions of enterprise KGs and explicitly refer to the industry perspective proposed by Noy et al. [68]. In that definition—co-authored by leaders of large-scale KG projects at eBay, Facebook, Google, IBM, and Microsoft—a KG is described as "objects of interest and connections between them". Ehrlinger, in [26], also summarizes several alternative definitions found in the literature.

2.4.2 Data Graph Models

As noted by Hogan et al. [39], KG construction fundamentally starts with imposing a graph abstraction on the data in order to obtain an initial data graph. Such a graph abstraction can be used to encode data by representing objects in a domain of interest as nodes and capturing the relationships between these entities as edges [5]. Researchers typically distinguish a small number of fundamental graph models, most notably *directed edge-labelled graphs* and *property graphs*, which are commonly treated as core abstractions (cf. [39]; [5]). Beyond these, additional graph types are frequently discussed in the context of KG research, including heterogeneous graphs [39] as well as named graphs, which allow grouping triples into identifiable graph partitions [11].

Edge-Labelled Graph. An *edge-labelled graph* is defined as a set of nodes together with a set of directed, labelled edges, where each edge is annotated with a label indicating the type of relationship it expresses [39]. In the following, we formalise the notion of an edge-labelled graph.

Definition 2 (Edge-labelled graph, adapted from [5])**.** An *edge-labelled graph* is a pair $G = (V, E)$ where:

1. V is a finite set of vertices (nodes),
2. $E \subseteq V \times Lab \times V$ is a finite set of labelled edges, where Lab denotes a set of edge labels.

Example 1 (Edge-labelled graph based on Figure 2)**.** Let $G = (V, E)$ denote the graph shown above. Then:

$$V = \{\textit{GIGA R1 WiFi}, \textit{GettingStartedDoc}\}$$

$$E = \{(\textit{GIGA R1 WiFi}, \textit{DESCRIBED\_IN}, \textit{GettingStartedDoc})\}$$

Figure 2 illustrates this idea using a simplified example from the Arduino product domain. A widely adopted and standardised data model built upon directed edge-labelled graphs is the Resource Description Framework (RDF), as recommended by the W3C [16]. RDF distinguishes several types of nodes, most notably Internationalized Resource Identifiers (IRIs) [22], which enable the global and unambiguous identification of entities across the Web [39].

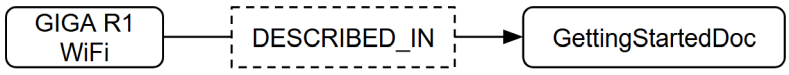


Figure 2: Edge-Labelled Graph Example

Property Graphs. *Property graphs* [39], also referred to as *labeled property graphs* [21], extend the concept of edge-labelled graphs by allowing both nodes and edges to carry labels as well as properties [39]. In edge-labelled graphs, the type of a relationship is expressed solely through the label on an edge, while class information for nodes has to be modelled indirectly. [39] and [5] point out that property graphs make this more explicit by allowing nodes to be labelled directly and by supporting labels for nodes and edges, as well as key–value properties that enable more expressive representations of structured data.

Definition 3. A property graph is a tuple $G = (V, E, \rho, \lambda, \sigma)$, where:

1. V is a finite set of vertices (nodes),
2. E is a finite set of edges such that $V \cap E = \emptyset$,
3. $\rho : E \to (V \times V)$ is a total function mapping each edge to its source and target nodes,
4. $\lambda : (V \cup E) \to Lab$ is a total function assigning a label to each node or edge, where Lab is a set of labels,
5. $\sigma : (V \cup E) \times Prop \to Val$ is a partial function assigning values to properties from a finite set $Prop$.

Example 2 (Property graph based on Figure 3)**.** Let $G = (V, E, \rho, \lambda, \sigma)$ be the property graph depicted above.

Vertices and edges. The vertex set is $V = \{n_1, n_2\}$ and the edge set is $E = \{e_1\}$.

Node identities. Node n_1 represents *GIGA R1 WiFi*, while n_2 represents *GettingStartedDoc*.

Relationship mapping. The edge e_1 connects n_1 to n_2, formally given by

$$\rho(e_1) = (n_1, n_2).$$

Labels. The labeling function assigns $\lambda(n_1) = \textit{Product}$, $\lambda(n_2) = \textit{Document}$, and $\lambda(e_1) = \textit{DESCRIBED\_IN}$.

Properties. The property function is defined as follows: $\sigma(n_1, \textit{category}) = \textit{Mega}$, $\sigma(n_2, \textit{title}) = \textit{Getting Started}$, $\sigma(n_2, \textit{url}) = \textit{https://www.arduino.cc/...}$, and $\sigma(e_1, \textit{relevance}) = 0.95$.

Figure 3 illustrates this idea using an example from the Arduino product domain.

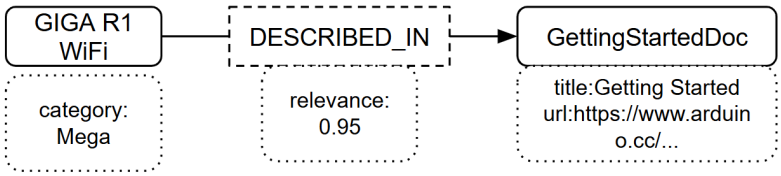


Figure 3: Property Graph Example

Property graphs are used widely [42], most prominently in popular graph databases such as Neo4j [39, 21]. In RDF, data is represented as subject–predicate–object triples, where the subject and object are nodes and the predicate defines a directed edge between them. The object can be either another node or a literal value. This model is fundamentally edge-centric, as the relationship (predicate) is the central organizing element. In contrast, the Labeled Property Graph model, which is natively used by Neo4j, allows both nodes and edges to carry arbitrary key–value pairs as properties. This makes LPGs particularly well-suited for applications requiring rich metadata. Unlike RDF, LPGs are generally node-centric, placing entities at the center of the data structure rather than the relationships between them [21].

2.4.3 Knowledge Graph Construction

KG construction refers to the process of identifying and organizing the elements that constitute a KG, either within a domain-specific or an open-domain setting.

Formally, it can be described as a mapping

$$f : D \times f_k(D) \rightarrow G,$$

where D denotes the set of data sources, $f_k(D)$ represents background knowledge about the data or target domain, and G is the resulting KG. The background knowledge may encode domain-specific information and is typically provided through predefined rules or learned representations. As noted in the literature, the construction process generally relies on such background knowledge to guide the extraction and structuring of graph elements [106]. Numerous surveys have investigated the problem of KG construction from textual data. Wu et al. [101] provide a comprehensive review of competitive tools and models for key KG construction subtasks, including named entity recognition and relation extraction. Other studies focus specifically on deep learning approaches for the joint extraction of entities and their relations, addressing the strong interdependence between these tasks [64, 73]. More recently, Zhu et al. [108] examine the emerging use of large language models for KG construction, highlighting their potential to unify multiple extraction steps within a single framework. Han et al. [37] distinguish three common approaches to KG construction: *manual construction*, where graphs are created through human annotation (e.g., Wikidata or UMLS); *rule-based construction*, which applies predefined rules and classical NLP pipelines to extract entities and relations (e.g., Freebase or ConceptNet); and *LLM-based construction*, which leverages large language models to automatically derive graph structures from text collections.

When addressing the problem of KG construction, several core information extraction tasks arise, most notably *Named Entity Recognition*, *Relation Extraction*, and *Triplet Extraction*. In the following, these concepts are briefly introduced and discussed individually.

Named Entity Recognition. Named Entity Recognition (NER) is a well-studied task in natural language processing that focuses on identifying and classifying entity mentions in text into predefined categories, such as persons, locations, or organizations [103]. As a core information extraction component, NER serves as a foundational building block for a wide range of downstream natural language applications [37]. Formally, Named Entity Recognition (NER) can be defined as a sequence labeling problem. Given an input text represented as a token sequence

$$X = (x_1, x_2, \ldots, x_n),$$

the objective of NER is to predict a corresponding sequence of entity labels

$$Y = (y_1, y_2, \ldots, y_n),$$

where each label $y_i \in \mathcal{L}$ denotes either a predefined entity category (e.g., PERSON, LOCATION, ORGANIZATION) or a non-entity class. The task is commonly modeled as learning a function

$$f_\theta : X \rightarrow Y,$$

parameterized by θ, which maximizes the conditional probability $P(Y \mid X)$ over labeled training data.

Relation Extraction. Similar to Named Entity Recognition, Relation Extraction (RE) is a well-established task in natural language processing that aims to identify and classify semantic relationships between entities mentioned in text. RE has been widely applied in downstream applications such as structured information retrieval, sentiment analysis, question answering, text summarization, and KG construction. By extracting relational information from unstructured text, RE transforms raw textual content into structured representations that explicitly capture semantic links between entities [19]. Formally, Relation Extraction (RE) aims to identify semantic relations between pairs of entities occurring in text. Given an input token sequence

$$X = (x_1, x_2, \ldots, x_n)$$

and a set of entity mentions

$$E = \{e_1, e_2, \ldots, e_m\},$$

the objective of RE is to predict a relation label

$$r \in \mathcal{R} \cup \{\text{NONE}\}$$

for a given entity pair (e_i, e_j), where $\mathcal{R}$ denotes a predefined set of relation types. The task is commonly formulated as a classification problem by learning a function

$$g_\theta : (X, e_i, e_j) \to r,$$

parameterized by θ, which maximizes the conditional probability $P(r \mid X, e_i, e_j)$.

Triplet Extraction. Triplet Extraction (TE) is a central task in Natural Language Processing (NLP) [64], as the semantic content of a sentence can often be represented through a set of subject–predicate–object triplets [71]. Such triplets provide a structured representation of relational information expressed in natural language.

Recent approaches to TE increasingly adopt an end-to-end modeling paradigm, in which the entire extraction process is learned jointly[71]. In this setting, models are trained to perform multiple sub-tasks simultaneously, including Named Entity Recognition [103], Entity Linking [4], and Relation Extraction [19], within a unified framework [71]. Formally, Triplet Extraction can be defined as a mapping from natural language text to a set of relational triples. Let T denote an input text sequence and let $\mathcal{E}$ be the set of entities mentioned in T. The task of Triplet Extraction consists in identifying a set

$$\mathcal{R}_T = \{(s, p, o) \mid s, o \in \mathcal{E},\ p \in \mathcal{P}\},$$

where s denotes a subject entity, o an object entity, and p a relation predicate from a predefined set of relations $\mathcal{P}$. Each triple (s, p, o) represents a relational fact expressed or implied in the text T. In end-to-end formulations, the extraction process

jointly performs entity identification, entity linking, and relation prediction, producing $\mathcal{R}_T$ directly from the input text without explicit intermediate representations.

Triplet Extraction can be addressed using LLMs by framing the task as a sequence generation problem. Given an input sentence, an LLM can be prompted to directly generate a set of subject–predicate–object triplets corresponding to the relational facts expressed in the text. In this end-to-end formulation, entity identification, linking, and relation prediction are performed implicitly within the model[71].

2.4.4 Entity Resolution in Knowledge Graph

In the context of KG construction, the question of whether different records refer to the same real-world entity may arise frequently. Information used to populate a KG is often collected from multiple documents or data sources and can be represented in heterogeneous and potentially inconsistent forms. For example, the same entity may be referenced using different spellings, abbreviations, or naming conventions across documents, while similar surface forms may, in other cases, denote distinct entities. When such information is transformed into nodes and relationships within a KG, these ambiguities can result in multiple nodes representing the same underlying entity or, conversely, in the unintended merging of distinct entities. Consequently, determining whether two records should be linked to a single node or represented as separate entities becomes a relevant consideration during KG construction. This issue is commonly addressed within the scope of *Entity Resolution (ER)*, a research area that has been studied for several decades across text processing, database systems, and, more recently, machine learning communities, employing a variety of methodologies such as rule-based and statistical approaches [48]. In the context of KG construction and integration, ER constitutes a fundamental component for identifying and linking entities that refer to the same real-world object across different KGs [69]. We therefore define Entity Resolution formally as follows.

Definition 4 (Entity Resolution)**.** Let $KG = (\mathcal{E}, \mathcal{A}, \mathcal{R}, \mathcal{L}, \mathcal{T})$ be a KG with entity set $\mathcal{E}$, and let $\mathcal{O}$ denote the set of real-world objects. We assume the existence of an unknown mapping $\phi : \mathcal{E} \to \mathcal{O}$ assigning each entity to the real-world object it represents. Entity Resolution is defined as the task of determining an equivalence relation $\equiv \subseteq \mathcal{E} \times \mathcal{E}$ such that for any $e_i, e_j \in \mathcal{E}$,

$$e_i \equiv e_j \iff \phi(e_i) = \phi(e_j).$$

One general approach that can be applied across different entity types involves the use of graph embeddings. In this setting, entities from KGs are represented in a low-dimensional vector space, which may help to reduce the effects of structural and semantic heterogeneity present in the original graphs. By capturing both topological structure and semantic relatedness within a geometric embedding space, such representations can subsequently be used as input features for machine learning algorithms [69].

2.4.5 Community Detection in Knowledge Graph

This chapter introduces the concept of community detection in knowlegde graphs and discusses its fundamental principles. Community detection is a specialized research area that focuses on identifying structural patterns and interactional regularities within graphs by grouping nodes into communities based on the information encoded in the graph structure. The problem of community detection has been extensively studied in graph analysis and has received considerable attention across multiple domains, including social network analysis [28, 50] and information retrieval.

A community is typically understood as a group of nodes that are more densely connected to each other than to the rest of the graph. One of the characteristic properties of graphs representing real-world systems is the presence of community structure, also referred to as clustering. In this context, vertices tend to be organized into groups such that nodes within the same group are connected by a relatively large number of edges, while connections between nodes belonging to different groups are comparatively sparse. These groups, commonly called communities, can be interpreted as semi-independent substructures within the graph, analogous to functional compartments in complex systems. The task of community detection is therefore concerned with partitioning a graph into such communities based on patterns of dense internal connectivity and sparse external connectivity. Community detection has been studied extensively in various disciplines, including sociology, biology, and computer science, where graph-based representations are frequently employed [28]. In the following, we formalize the notion of community detection.

Definition 5 (Hierarchical Community Detection)**.** Let $G = (V, E)$ be a graph. Hierarchical community detection consists in determining a sequence of partitions

$$\mathcal{C}^{(1)}, \mathcal{C}^{(2)}, \ldots, \mathcal{C}^{(L)}$$

of the vertex set V, where each partition represents a level of granularity and communities at higher levels are formed by unions of communities at lower levels.

A variety of methods have been proposed to address the problem of community detection in graphs, differing in their underlying assumptions and optimization objectives. One widely studied class of approaches is based on the concept of modularity, which assesses the quality of a partition by comparing the density of edges within communities to that expected under a null model [44]. Community detection is formulated in this setting as an optimization problem that seeks to maximize the modularity score.

Among modularity-based approaches, the Louvain method [10] is a prominent example due to its computational efficiency and scalability. The algorithm follows an iterative two-phase procedure that alternates between local modularity optimization and the aggregation of communities into a reduced graph. The Leiden algorithm [95] extends this approach by addressing limitations of Louvain, in particular

by ensuring that detected communities are internally well connected while preserving efficiency.

In addition to modularity-based methods, spectral clustering approaches constitute an important family of techniques for community detection. These methods exploit the spectral properties of the graph, typically derived from the graph Laplacian, to partition nodes based on low-dimensional representations obtained from eigenvalue decompositions. Spectral methods are particularly suited for capturing complex structural patterns in graphs and are widely used in graph partitioning and network analysis.

2.5 Knowledge Graph–Enhanced RAG

In recent years, the extension of RAG with graph-based structures has emerged as an active area of research aimed at improving retrieval and context integration. When combining RAG with graphs the literature employs a variety of terms, including *KG–RAG* [83], *KG–Guided RAG* [107], and *GraphRAG* [36, 37, 102, 74, 24, 100]. Despite differences in terminology and implementation, these approaches generally share the idea of augmenting RAG systems with graph-derived structure, either in the form of explicit KGs or graph-structured document representations.

Within this research direction, Zhu et al. [107] proposed a Knowledge Graph–Guided RAG framework, referred to as *KG2RAG*. The approach performs an offline preprocessing step in which documents are segmented into chunks and explicitly linked to a given KG, thereby establishing associations that capture fact-level relationships between text segments. During retrieval, KG2RAG combines semantic-based chunk retrieval with a graph-guided expansion strategy, enabling the system to incorporate relational context derived from the KG into the retrieved evidence.

A prominent and widely referenced contribution within this research direction was introduced by Microsoft Research in 2024, which proposed a community-centric GraphRAG framework that organizes document collections into graph-based communities to guide retrieval and summarization [24].

2.5.1 Related Techniques

This subsection briefly introduces techniques related to KG-enhanced RAG that address question answering with external knowledge sources. The presented approaches provide contextual background and help position KG-enhanced RAG within the broader landscape of knowledge-based and retrieval-augmented methods.

KBQA. Knowledge Base Question Answering (KBQA) is a core task in natural language processing that aims to answer user queries by leveraging external knowledge bases [29]. Typical objectives of KBQA include fact verification, enhanced retrieval of relevant passages, and improved text understanding. Prior survey work

commonly distinguishes two main categories of KBQA approaches: information retrieval–based methods and semantic parsing–based methods.

Information retrieval–based KBQA methods operate by retrieving graph elements related to a given query from a KG and using the retrieved information to support answer generation. In this regard, KBQA and GraphRAG are closely related, as IR-based KBQA methods can be viewed as a specialized subset of GraphRAG approaches that focus on question answering as a downstream application [75].

Graph Neural Networks. Graph Neural Networks (GNNs) are deep learning models designed to learn representations from graph-structured data by iteratively aggregating information from neighboring nodes. Classical architectures such as GCN, GAT, and GraphSAGE follow a message-passing paradigm in which node representations are updated based on local graph neighborhoods using permutation-invariant aggregation functions. After multiple layers of message passing, readout functions can be applied to derive graph-level representations. In the context of GraphRAG, GNNs can be employed to encode graph structures and support retrieval or modeling of relational information during the generation process [75].

2.5.2 GraphRAG

GraphRAG refers to a class of RAG approaches that extend conventional RAG systems by incorporating graph-structured data into the retrieval process. In contrast to standard RAG, which predominantly relies on semantic or lexical similarity search over unstructured text corpora, GraphRAG aims to exploit relational information encoded in graphs to retrieve more structured and contextually connected evidence [36].

The GraphRAG paradigm was explicitly introduced as an architectural extension of RAG in which a structured KG is embedded between the retrieval and generation stages [37]. In this setting, entity–relation graphs are constructed from retrieved passages and subsequently organized into higher-level semantic structures, such as communities, often supported by LLM-based summarization mechanisms [62].

Equipping RAG with graph structures requires adaptations of its core components, in particular the retriever and the generator, to enable seamless integration of graph-based representations. Depending on the specific realization, GraphRAG systems may leverage graph-based machine learning techniques, such as graph neural networks, as well as graph and network analysis methods, including graph traversal strategies and community detection, to capture relational dependencies that are not explicitly accessible through text similarity alone [37].

From a broader perspective, GraphRAG can be viewed as a branch of RAG that retrieves relevant information from graph-structured data sources, such as KGs or document graphs, rather than directly from raw text collections. In this setting, retrieval operates over graph elements, including nodes, edges, or subgraphs, while

generation is conditioned on the retrieved structured context. Compared to text-based RAG, GraphRAG explicitly accounts for relationships between information units and incorporates structural signals as an additional source of knowledge beyond textual content. Moreover, during graph construction, raw text may be filtered, aggregated, or summarized before being embedded into the graph, resulting in a more refined and organized representation of the underlying information [75].

Han et al. [36] distinguish three representative variants of GraphRAG. The first variant, referred to as *knowledge-graph-based GraphRAG*, constructs an explicit KG from text and performs retrieval solely over graph elements such as entities and relations [83]. The second variant, *community-based GraphRAG*, extends this approach by additionally leveraging hierarchical community structures within the graph, enabling retrieval at different levels of abstraction [23]. The third variant, often termed *text-based GraphRAG*, uses graph structures to guide retrieval while ultimately returning original text chunks as context for generation, as exemplified by approaches such as HippoRAG [34].

Let $\mathcal{C} = \{c_1, \ldots, c_n\}$ denote a set of text chunks obtained from a document collection. From these chunks, a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is constructed, where nodes $v \in \mathcal{V}$ represent entities, concepts, or aggregated textual units extracted from $\mathcal{C}$, and edges $e \in \mathcal{E}$ encode semantic, relational, or structural dependencies inferred from the underlying text. As a result, each graph element is grounded in one or more text chunks, but the chunks themselves are no longer the primary retrieval units. Given a user query q, GraphRAG aims to identify a relevant subgraph $\mathcal{G}^* \subseteq \mathcal{G}$ that maximizes the conditional likelihood of producing a correct answer. Formally, the answer generation objective can be expressed as

$$a^* = \arg\max_{a \in \mathcal{A}} p(a \mid q, \mathcal{G}),$$

where $\mathcal{A}$ denotes the space of possible responses. Due to the combinatorial number of candidate subgraphs, this objective is commonly approximated by decomposing the conditional distribution into a graph retrieval component and a generation component,

$$p(a \mid q, \mathcal{G}) \approx p_\phi(a \mid q, \mathcal{G}^*)\, p_\theta(\mathcal{G}^* \mid q, \mathcal{G}),$$

where p_θ denotes a graph retriever responsible for selecting the most relevant subgraph $\mathcal{G}^*$, and p_ϕ denotes a generator conditioned on the retrieved graph context [75]. In the current literature, the distinction between knowledge-graph–enhanced RAG and GraphRAG is not defined in a fully consistent or universally accepted manner. Some scientific sources interpret knowledge-graph–enhanced RAG as a subcategory of GraphRAG [36], while others explicitly refer to the extension of RAG with knowledge graphs as KG-RAG [83]. This terminological ambiguity is further reflected in practical implementations, where both terms are often used interchangeably. For instance, Neo4j defines GraphRAG broadly as "GraphRAG is Retrieval Augmented Generation (RAG) using a Knowledge Graph" [66], whereas Microsoft characterizes GraphRAG as "a technique for richly understanding text datasets by

combining text extraction, network analysis, and LLM prompting and summarization into a single end-to-end system" [61]. Similarly, IBM describes GraphRAG as "an advanced version of retrieval-augmented generation (RAG) that incorporates graph-structured data, such as knowledge graphs (KGs)" [41]. In this thesis, we use the term *KG-enhanced RAG* as an umbrella term to denote retrieval-augmented generation approaches that integrate knowledge graphs at any stage of the retrieval or generation pipeline. This includes methods that are referred to as *GraphRAG*, *KG-RAG*, or related variants in the literature.

2.5.3 KG-enhanced RAG Retrievers

This subsubsection outlines different categories of graph-based retrieval approaches, including neural, language-model-driven, non-parametric, and KG-enhanced RAG-based retrievers. Each category represents a distinct strategy for accessing and exploiting structured KGs during retrieval. Let $G = (V, E)$ denote a graph-based knowledge source and q a natural language query. A retriever in KG-enhanced RAG defines a function

$$\mathcal{R} : (q, G) \mapsto G_q \subseteq G,$$

which selects a query-relevant subgraph G_q used for downstream generation. Existing approaches can be categorized according to the modeling assumptions of $\mathcal{R}$ into non-parametric, language-model-based, and graph-neural-network-based retrievers [75].

Non-parametric Retrievers. Non-parametric retrievers rely on heuristic graph search procedures rather than learned models. Formally, retrieval is defined via deterministic traversal operators such as k-hop expansion, shortest-path search, or Steiner tree approximations:

$$G_q = \mathrm{Search}(G, V_q),$$

where $V_q \subseteq V$ denotes query-linked entities. These methods offer high efficiency but limited semantic generalization.

LM-based Retrievers. LM-based retrievers parameterize $\mathcal{R}$ using a language model that maps queries to graph elements:

$$p_\theta(e \mid q) \quad \text{or} \quad p_\theta(p \mid q),$$

where $e \in V$ and $p \in E$. Retrieval is performed by ranking entities, relations, or paths according to model scores, enabling semantic matching between natural language queries and graph structure.

GNN-based Retrievers. GNN-based retrievers first encode graph nodes using message passing to obtain embeddings $\mathbf{h}_v \in \mathbb{R}^d$ for $v \in V$. Relevant subgraphs are selected by scoring nodes or paths based on similarity to a query representation $\mathbf{h}_q$, typically via

$$\text{score}(v, q) = \cos(\mathbf{h}_v, \mathbf{h}_q).$$

This class of methods explicitly models relational dependencies but incurs higher computational cost.

3 Methodology and System Design

All software components developed for this work, including the RAG pipeline, the KG construction scripts, and the assessment framework, have been published as open-source and are available in the accompanying GitHub repository (see Appendix 5). The empirical part of this thesis is based on a dataset of approximately 200 publicly available technical documents from the Arduino ecosystem [7]. These documents, provided under a permissive Creative Commons license, include product manuals, datasheets,tutorials, and setup guides, which serve as a reliable basis for information retrieval and knowledge extraction.

3.1 Dataset and Data Sources

Arduino is an open-source hardware and software ecosystem designed to make the development of interactive electronic systems accessible to a broad audience. It combines microcontroller-based development boards with a unified programming environment and a large collection of libraries, documentation, and community resources. Because Arduino significantly lowers the entry barrier to embedded programming, it has become widely used in education, prototyping, hobbyist projects, and early-stage research.

Arduino offers a diverse portfolio of products targeting different application domains. The classical category comprises microcontroller boards such as the *Arduino Uno*, *Arduino Nano*, and *Arduino Mega*, which provide basic digital and analog input/output capabilities. These boards are typically used in introductory electronics courses, rapid prototyping, and small-scale hardware experiments. More advanced and high-performance devices include the *Portenta H7* and the *GIGA R1 WiFi*, which provide substantially more computational power, improved connectivity, and support for demanding tasks such as real-time processing or embedded machine learning.

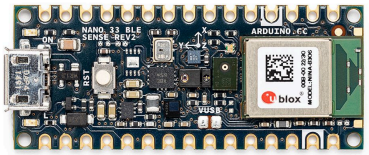
Another major product line consists of Internet-of-Things (IoT) boards with integrated wireless communication modules. Examples include the *Nano 33 IoT* with WiFi and BLE support, and the *MKR WAN 1310*, which enables long-range, low-power communication via LoRa. These boards are often used in smart home systems, environmental monitoring, remote sensing, and cloud-connected applications. Additionally, Arduino provides a range of peripheral modules and shields,

such as motor controllers, sensor boards, and communication shields, which extend the functionality of the core devices and allow users to assemble complete embedded solutions. Educational kits and thematic bundles further support beginners by offering curated hardware and step-by-step instructional material.

For this thesis, a comprehensive collection of documentation covering these product families was gathered. Most Arduino products are accompanied by detailed *datasheets* available in PDF format. These documents follow a relatively consistent structure across product lines, typically covering device overviews, pinout diagrams, electrical characteristics, communication interfaces, system descriptions, and hardware specifications. Their consistent organisation makes them particularly suitable for structured extraction and knowledge-graph construction.

In addition to datasheets, many products also provide *tutorials*, *getting-started guides*, and *example project descriptions*, which were likewise extracted in PDF format. These documents contain procedural explanations and narrative descriptions that complement the technical information found in datasheets. Altogether, approximately 200 documents were collected, combining both structured and unstructured material across a wide variety of Arduino products. This corpus forms the dataset used for constructing the KG and for evaluating the retrieval and generation methods in this thesis.

In the following, a snapshot of Arduino product datasheets is presented as an illustrative example.



Description

The Arduino® Nano 33 BLE Sense Rev2* is a miniature sized module containing a NINA B306 module, based on Nordic nRF52480 and containing an Arm® Cortex®-M4F. The BMI270 and BMM150 jointly provide a 9 axis IMU. The module can either be mounted as a DIP component (when mounting pin headers), or as a SMT component, directly soldering it via the castellated pads.

1.1 Processor

The Main Processor is an Arm® Cortex®-M4F running at up to 64 MHz. Most of its pins are connected to the external headers, however some are reserved for internal communication with the wireless module and the onboard internal I2C peripherals (IMU and Crypto). **NOTE:** *As opposed to other Arduino Nano boards, pins A4 and A5 have an internal pull up and default to be used as an I2C Bus so usage as analog inputs is not recommended.*

1.2 IMU

The Nano 33 BLE Sense Rev2 provides IMU capabilities with 9-axis, by combination of the BMI270 and BMM150 ICs. The BMI270 includes both a three axis gyroscope as well as a three axis accelerometer, while the BMM150 is capable of sensing magnetic field variations in all three dimensions. The information obtained can be used for measuring raw movement parameters as well as for machine learning.

1.3 USB

Pin	Function	Type	Description
1	VUSB	Power	Power Supply Input. If board is powered via VUSB from header this is an Output (1)
2	ID	Analog	Selects Host/Device functionality

3.2 Pre-Retrieval and Data Preparation

This subsection describes the data preparation steps performed prior to pipeline construction, including document parsing, chunking, index construction and metadata assignment. These steps establish the structural and semantic foundations required for all subsequent retrieval and augmentation processes.

3.2.1 Document Parsing and Chunking

A central methodological component of this thesis is the careful preparation and segmentation of the Arduino documentation corpus. Since the effectiveness of both RAG and KG-enhanced RAG depends heavily on the quality of the underlying text units, substantial effort was devoted to designing a robust and domain-sensitive document processing pipeline.

At the beginning of this work, several manual and heuristic chunking strategies were explored, including segmentation based on a fixed number of tokens. Although such approaches are widely used in simple RAG systems, they proved unsuitable for the technical documentation considered here. Fixed-size chunking often cuts through semantically coherent sections, merges unrelated content, or separates crucial contextual information (e.g., diagrams, pin descriptions, electrical specifications) from the text that explains them. This leads to segments that are internally incoherent, difficult to embed reliably, and less informative for retrieval tasks. As a consequence, the retrieval model frequently returned irrelevant or misleading chunks, and the generation model struggled to maintain factual consistency. These shortcomings motivated the adoption of a more structure-aware approach.

For this purpose, the *Docling* [20] framework was selected as the foundation of the document processing pipeline. Docling performs an initial transformation of each PDF into an internal, layout-informed document representation. Instead of treating the PDF as flat text, Docling reconstructs higher-level structural elements such as headings, textbfs, tables, lists, and hierarchical section boundaries. It also provides robust OCR support for scanned or partially image-based documents. This enriched intermediate representation ensures that the logical structure of the original document is preserved and can be leveraged during segmentation.

A key component used in this thesis is Docling's *HybridChunker*. In contrast to naive token-based splitting, the hybrid method integrates two complementary signals: explicit document structure extracted by Docling's conversion pipeline, as well as semantic and lexical constraints derived from token-level analysis. As a result, chunk boundaries tend to align with meaningful structural units, such as section headings or cohesive textbfs, while still remaining compatible with the token limitations of modern embedding models. Given that Arduino datasheets and tutorials follow relatively consistent organisational patterns—covering, for example, device overviews, pinout descriptions, electrical characteristics, and usage instructions—the hybrid chunker preserves these natural units and produces segments that are semantically coherent and contextually self-contained.

To further improve chunk quality, a customised text-cleaning and filtering pipeline was developed. Technical PDFs often contain noise such as repeated headers and footers, table remnants, hyperlink blocks, indices, revision histories, or other formatting artefacts. Such elements can distort embedding vectors and degrade retrieval accuracy. Therefore, heuristics were implemented to remove irrelevant material, normalise whitespace, join hyphenated line breaks, filter sections dominated by URLs or link tables, and eliminate headings that contribute little semantic value. Chunks with insufficient textual density or low information content were discarded. Each remaining chunk was enriched with metadata describing the product category, product name, and associated section heading, ensuring that downstream retrieval processes can make use of structured contextual information. Figure 4 provides a snapshot of how the hybrid Docling chunker transforms document sections into semantically aligned chunks.

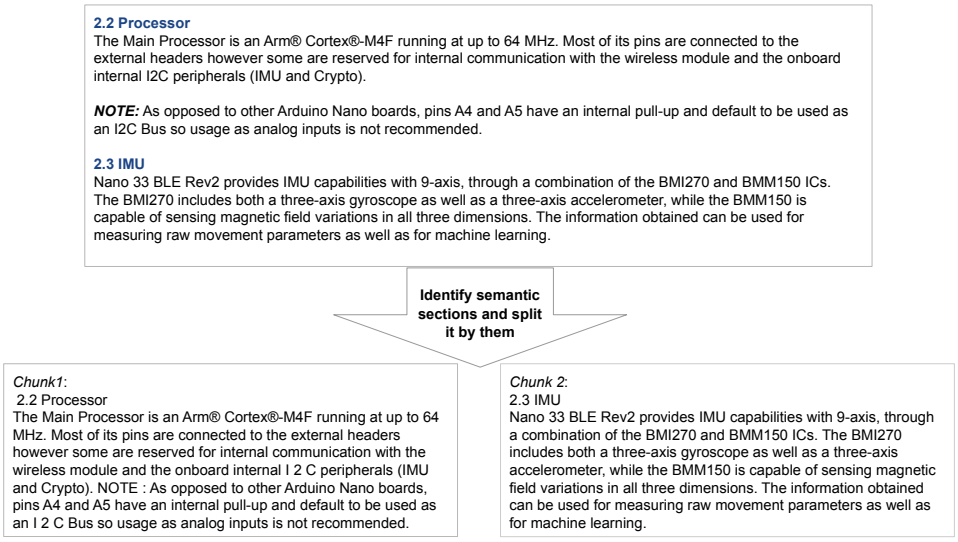


Figure 4: Illustration of the Chunking Strategy

Overall, the pipeline developed for this thesis combines Docling's layout-preserving PDF conversion, hybrid chunking, and domain-specific filtering to produce high-quality text segments suited for both RAG and KG-enhanced retrieval. The processing of approximately 200 Arduino documents required extensive experimentation to calibrate chunk sizes, validate semantic coherence, and ensure reproducibility. Compared to manual or fixed-size approaches, Docling's structurally informed segmentation proved significantly more effective, producing chunks that align with the intentional organisation of technical documentation and thereby enabling a more reliable and interpretable assessment of retrieval methods. This step constitutes a substantial methodological contribution of the thesis and provides a solid foundation for the empirical analyses presented in subsequent chapters.

3.2.2 Index Construction in Neo4j

In this step, the retrieval backbone of the baseline RAG system is constructed by persisting document chunks in a Neo4j graph database and preparing both dense and sparse indices. First, all chunk files are loaded by recursively scanning a directory tree for JSONL files. Each JSON line represents an individual document chunk and is parsed into a *Document* object, where the chunk text forms the content and metadata fields such as product, product category, and file name are attached. Chunks without textual content are discarded to ensure that only retrievable passages are indexed.

Let $D = \{d_1, \ldots, d_n\}$ denote the resulting set of document chunks, where each chunk d_i is associated with textual content t_i and metadata m_i. Dense vector representations are computed using an OpenAI embedding model, yielding embeddings $\mathbf{x}_i \in \mathbb{R}^d$. Each chunk is then persisted as a node with label Chunk in Neo4j, storing the raw text in the text property and the corresponding embedding vector in the embedding property. Dense vector representations are computed using the OpenAI text-embedding-3-small model, producing embeddings $\mathbf{x}_i \in \mathbb{R}^{1536}$. A Neo4j vector index is created via Neo4jVector, enabling efficient similarity-based retrieval over the embedding space.

By persisting embeddings in the graph database, the retrieval layer avoids repeated embedding computation and supports efficient reuse across multiple assessment runs. At this stage, no retrieval queries are executed; the process is limited to index construction and data preparation for subsequent retrieval and generation phases. In addition to dense retrieval, a separate setup step creates a full-text index chunk_text_ft over the chunk text property. Before creation, the script checks whether the index already exists to prevent redundant operations. This index enables keyword-based retrieval using Neo4j's full-text search capabilities and is later combined with vector-based similarity search in a hybrid retrieval strategy.

3.3 Baseline Advanced Retrieval-Augmented Architecture

This subsection describes the implementation of a baseline RAG pipeline without the use of a KG. It outlines the construction of retrieval indices, the design of a naive advanced RAG pipeline, and the integration of post-retrieval processing and reranking components.

3.3.1 Building Naive Advanced Retrieval-Augmented Pipeline

To enable a systematic comparison with the proposed KG-enhanced RAG approach, a conventional RAG pipeline was implemented as a baseline, starting with a naive variant.

Given an input question $q \in \mathcal{Q}$, the dense retrieval step performs a similarity-based search over the precomputed vector index to obtain the top-k most similar

document chunks. Formally, let $D = \{d_1, \ldots, d_n\}$ denote the indexed set of document chunks with embedding representations $\mathbf{x}_i \in \mathbb{R}^d$. The retrieval function returns the top-k most similar chunks

$$D_k(q) = \mathrm{TopK}_{d_i \in D}\ \mathrm{sim}(\mathbf{e}(q), \mathbf{x}_i),$$

where $\mathbf{e}(q) \in \mathbb{R}^d$ denotes the embedding of the query and $\mathrm{sim}(\cdot, \cdot)$ represents the similarity function used by the vector index.

The retrieved chunks are concatenated verbatim to form a flat textual context $C(q)$, which is passed unchanged to the language model. No filtering, reranking, or structural aggregation of the retrieved passages is applied. Consequently, the augmentation step can be formalized as

$$C(q) = \bigoplus_{d_i \in D_k(q)} t_i,$$

where t_i denotes the textual content of chunk d_i and $\oplus$ represents string concatenation.

In the generation step, the augmented prompt consists of a fixed instruction template, the concatenated context $C(q)$, and the original question q. A chat-based large language model is then invoked to generate an answer $a \in \mathcal{A}$:

$$a = \mathrm{LLM}(q, C(q)).$$

For evaluation and analysis purposes, the pipeline logs both the generated answer and the set of retrieved context chunks for each question. This enables post-hoc inspection of retrieval quality and comparison with more advanced RAG variants.

3.3.2 Building Post-Retrieval and Reranking

After constructing the vector store, the design of the advanced retrieval pipeline builds upon the architecture proposed in recent RAG research, particularly the categorisation of Pre-Retrieval, Retrieval, and Post-Retrieval components as outlined in [32]. Following this conceptual structure, an Advanced RAG system was implemented. The goal of this extended pipeline is to overcome the limitations of naive RAG pipeline.

Advanced RAG with Hybrid Retrieval. Algorithm 1 illustrates the implemented advanced hybrid RAG pipeline, which combines LLM-based query transformation, sparse and dense retrieval, score fusion, and LLM-based reranking. In the pre-retrieval stage, the original user query q is first rewritten by an LLM into a more concise and retrieval-oriented formulation q_{rew}. In addition, a small set of representative keywords K is extracted. This step aims to improve both lexical and semantic coverage in the subsequent retrieval phase. Retrieval is performed via two complementary channels. Sparse retrieval queries a full-text index using a Boolean query

Algorithm 1: Advanced Hybrid RAG: Pre-retrieval, sparse+dense retrieval, fusion, and LLM reranking

Input : User query q, full-text index $\mathcal{I}_{ft}$, vector index $\mathcal{I}_{vec}$, embedding function $e(\cdot)$, LLMs LLM_{router} and LLM_{rerank}, mixing weight $\alpha \in [0, 1]$

Output : Selected documents D_{sel} and concatenated context C

function PreRetrieval(q):
    $q_{orig} \leftarrow q$
    $q_{rew} \leftarrow \text{LLM}_{router}$.rewrite($q$)
    $K \leftarrow \text{LLM}_{router}$.extractKeywords($q_{rew}$)
    return $\{q_{orig}, q_{rew}, K\}$

function RetrieveSparse(pre, k_s):
    Construct full-text query r_{ft} from pre.q_{rew} and pre.K
    $S \leftarrow \mathcal{I}_{ft}$.FulltextSearch($r_{ft}, k_s$)
    return S

function RetrieveDense(pre, k_d):
    $\mathbf{q} \leftarrow e(\text{pre}.q_{rew})$
    $V \leftarrow \mathcal{I}_{vec}$.VectorSearch($\mathbf{q}, k_d$)
    return V

function MinMaxNorm($\{s(d)\}_{d \in A}$):
    $m \leftarrow \min_{d \in A} s(d)$,;
    ;
    $M \leftarrow \max_{d \in A} s(d)$
    if $M = m$ **then**
        return $\hat{s}(d) = 1 \ \ \forall d \in A$
    return $\hat{s}(d) = \dfrac{s(d) - m}{M - m} \ \ \forall d \in A$

function FuseHybrid(S, V, α):
    Compute $\hat{s}_s$ by MinMaxNorm over sparse scores in S
    Compute $\hat{s}_d$ by MinMaxNorm over dense scores in V
    $U \leftarrow$ union of documents from S and V by chunk identifier
    foreach $d \in U$ **do**
        $\hat{s}_s(d) \leftarrow 0$ if $d \notin S$;;
        ;
        $\hat{s}_d(d) \leftarrow 0$ if $d \notin V$
        $s_{hyb}(d) \leftarrow \alpha \cdot \hat{s}_d(d) + (1 - \alpha) \cdot \hat{s}_s(d)$
    Sort U by $s_{hyb}(d)$ in descending order
    return U

function PostRetrieval(pre, U, k_{top}):
    if U *is empty* **then**
        return $(\emptyset, \emptyset)$
    Construct candidate list B from top-25 docs in U with indices and metadata
    $I \leftarrow \text{LLM}_{rerank}$.selectTopIndices(pre.$q_{rew}, B, k_{top}$)
    if I *is empty* **then**
        $I \leftarrow \{0, 1, \ldots, k_{top} - 1\}$
    $D_{sel} \leftarrow \{U[i] \mid i \in I\}$
    $C \leftarrow$ concatenate texts (with metadata) of D_{sel}
    return (D_{sel}, C)

constructed from q_{rew} and the extracted keywords K, producing a set of candidate chunks with lexical relevance scores.

As part of Algorithm 1, dense retrieval embeds the rewritten query into the same vector space as the document chunks and retrieves semantically similar candidates from a vector index. Since sparse and dense retrieval scores are not directly comparable, both score sets are independently normalized using Min–Max normalization. The candidates from both channels are merged by their chunk identifiers, and a hybrid relevance score is computed as a weighted linear combination of normalized dense and sparse scores. This fusion step balances semantic similarity and exact term matching through the parameter α.

To further refine the candidate set, an LLM-based reranking step is applied. The top-ranked hybrid candidates are presented to an LLM, which selects the indices of the most relevant chunks with respect to the rewritten query. If the LLM does not return a valid selection, a deterministic fallback strategy selects the top-ranked candidates. Finally, the selected chunks are concatenated into a single context string, including relevant metadata. A separate answer-generation LLM produces the final answer conditioned on the original query and the fused context, while being explicitly constrained to rely only on the provided information.

Advanced RAG with Sparse and Dense Retrieval. In addition to the hybrid variant described in Algorithm 1, structurally equivalent pipelines were implemented for sparse-only and dense-only retrieval. These variants reuse the same pre-retrieval and post-retrieval components, while restricting the retrieval stage to a single channel. This design ensures a controlled comparison between sparse, dense, and hybrid retrieval strategies.

3.4 Knowledge Graph Construction

The construction of the KG in this work follows an LLM-based paradigm for schema induction and triple extraction. This design choice is motivated by the heterogeneous and unstructured nature of the Arduino documentation, which spans product descriptions, hardware components, configuration parameters, and procedural instructions across a wide range of devices and use cases.

In such a setting, specifying a fixed ontology in advance is impractical. A highly constrained schema risks omitting relevant concepts, while an overly generic schema reduces expressiveness and limits the usefulness of the resulting graph for downstream retrieval and reasoning tasks. Consequently, the graph schema is not predefined but instead induced dynamically from the document collection itself.

To address this challenge, both schema discovery and relational triple extraction are delegated to large language models. LLMs are capable of inferring entity types and relation predicates directly from natural language text, enabling flexible and adaptive KG construction without requiring manual ontology engineering. Within this LLM-based framework, multiple KG construction pipelines were implemented

and evaluated. All approaches share the same high-level objective—extracting entities and relations from text and persisting them as nodes and edges in a Neo4j graph database—but differ in their degree of structural guidance and control over intermediate extraction steps. The following sections describe these pipelines in detail.

3.4.1 Construction with SimpleKGPipeline

The first method is based on Neo4j GraphRAG package [65], in particular the SimpleKGPipeline and the SchemaFromTextExtractor components. The SchemaFromTextExtractor uses an LLM to infer a candidate graph schema from representative text samples. Given an input passage, it prompts the model to propose node types, relation types, and relevant properties, effectively performing schema induction as a text-to-structure transformation. This step decouples schema discovery from the subsequent large-scale extraction: instead of fixing the ontology manually, the system learns an initial schema from the corpus itself, which can then be refined or constrained by the practitioner.

The SimpleKGPipeline was evaluated in two configurations that differ in the degree of schema guidance used during extraction. Both variants rely on Neo4j's experimental GraphRAG toolchain and use an LLM to transform natural-language chunks into graph-structured representations, including entity nodes, relationship types, and associated metadata. A snapshot of the KG generated through the LangChain-based construction pipeline is presented in Figure 5.

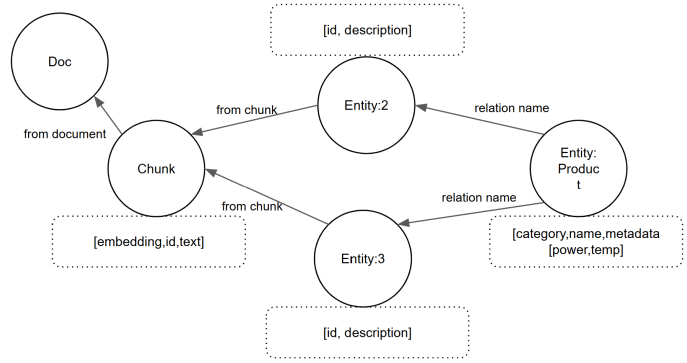


Figure 5: Snapshot of the Knowledge Graph Generated Using the SimpleKGPipeline

In the first configuration, the pipeline operates without an explicitly supplied schema. Each JSONL row is parsed into a text segment together with metadata describing the originating file, product category, and chunk identifier. These metadata fields are passed directly to the pipeline and later materialized as *Document* and *Chunk* nodes in Neo4j. The LLM processes each text segment independently and generates candidate entities and relations using the default extraction logic of the pipeline. This setting is suited for scenarios in which no prior assumptions about

the domain ontology are imposed. Embeddings for the resulting nodes are computed using the *text-embedding-3-small* model and stored within the Neo4j instance, enabling downstream vector-based retrieval in the GraphRAG workflow. The implementation processes all JSONL files asynchronously, ensuring that extraction is scalable across hundreds of text segments. Importantly, although an LLM-based schema extraction component was evaluated in preliminary experiments, no automatically induced or predefined schema is applied in this configuration; entity and relation extraction therefore remains fully schema-light and unconstrained.

3.4.2 Construction with LLMGraphTransformer

The second method relies on the LLMGraphTransformer from the LangChain experimental module [55]. Conceptually, this component treats graph construction as a document-to-graph transformation: it accepts Document objects (with text and metadata) and returns a set of GraphDocument objects, each containing explicit Node and Relationship structures. Internally, the transformer uses a prompt to instruct an LLM to identify entities, assign types, and link them via directed relations. The output is not yet stored in the database; instead, it is represented in an intermediate, library-agnostic graph format. This design has two advantages. First, it offers fine-grained control over how entities and relations are mapped into Neo4j. After running the LLMGraphTransformer, the user can iterate over the extracted nodes and edges, normalize labels, enrich properties, or introduce additional constraints before executing Cypher queries. Second, because the input is standard LangChain Document objects, the method integrates naturally with existing RAG pipelines that already use LangChain for chunking, embedding, and retrieval. In this thesis, the transformer is used to convert each text chunk into a local graph fragment, which is subsequently merged into a global KG and linked back to the originating chunk nodes.

The graph-construction workflow implements concrete processing steps used to transform chunked text into a structured Neo4j property graph, without repeating the conceptual description provided earlier. Each document receives metadata such as the filename, a stable chunk identifier, and selected preprocessing attributes. This metadata ensures that all later graph structures remain traceable to their textual origin and that each chunk can be reliably reconstructed as the source context of extracted entities and relations.

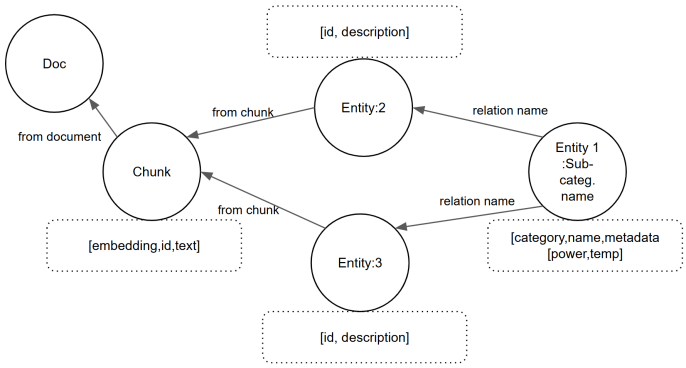


Figure 6: Snapshot of the Knowledge Graph Generated Using the LLMGraphTransformer

The documents are processed by the *LLMGraphTransformer*, which uses the OpenAI model (*gpt-4o-mini*) to predict entities, assign semantic types, and infer directed relations. The transformer returns these predictions as *GraphDocument* objects containing nodes, properties, and relationships. Because the transformer itself performs no database operations, this intermediate representation allows controlled inspection and normalization before persistence.

Each extracted graph fragment is inserted into Neo4j in a structured manner. A *Document* node (identified by the source filename) and a *Chunk* node (identified by the chunk ID) serve as contextual anchors. Entities predicted by the LLM are merged globally using their generated identifiers, ensuring that repeated mentions across multiple chunks resolve to a single node when appropriate. Each entity is linked to the corresponding chunk via *MENTIONS*. Predicted relations between entities are added using *MERGE* to avoid duplicates while retaining any generated properties. Uniqueness constraints on *Document*, *Chunk*, and *Entity* identifiers are enforced at the database level to ensure structural consistency and prevent duplicate node creation. A snapshot of the KG generated through the LangChain-based construction pipeline is presented in Figure 6.

3.4.3 Construction with SimpleLLMPathExtractor

The third KG construction method is based on the property-graph functionality provided by LlamaIndex [57], specifically the *SimpleLLMPathExtractor*. In contrast to approaches that focus on extracting isolated subject–predicate–object triples, this method conceptualizes knowledge extraction as the identification of relational paths within text. An LLM is prompted to detect sequences of semantically connected entities and relations, which are subsequently interpreted as paths in a property graph.

The *SimpleLLMPathExtractor* operates on individual text chunks and instructs the LLM to generate structured representations consisting of entities, relation types, and directed connections between them. These representations may capture multi-step

relational patterns, such as *Product* → *hasComponent* → *Sensor* → *measures* → *Quantity*, thereby encoding higher-level structural information that goes beyond single-hop relations. The extracted paths are internally mapped to LlamaIndex's *PropertyGraphIndex*, which serves as an intermediate graph abstraction.

In the implemented setup, the extractor is applied to the same chunked Arduino documentation used in the previous pipelines. Each text chunk is enriched with metadata, including product name, product category, chunk identifier, and source file information. The *SimpleLLMPathExtractor* processes each chunk independently and generates a set of candidate relational paths, constrained by a maximum number of extracted paths per chunk. This design makes the approach particularly suitable for capturing hierarchical, functional, or compositional structures commonly found in technical documentation.

The resulting graph is not stored in an ad-hoc format. Instead, the *PropertyGraphIndex* integrates both the extracted relational paths and dense vector representations of the original text chunks, computed using the *text-embedding-3-small* embedding model. This dual representation enables the reuse of the same graph for both symbolic graph traversal and vector-based retrieval in subsequent RAG and GraphRAG experiments. The final graph is persisted directly to Neo4j via the *Neo4jPGStore*, ensuring that all nodes and relationships remain explicitly linked to their originating text chunks and documents. This guarantees full traceability during evaluation, as each graph element can be traced back to its textual source.

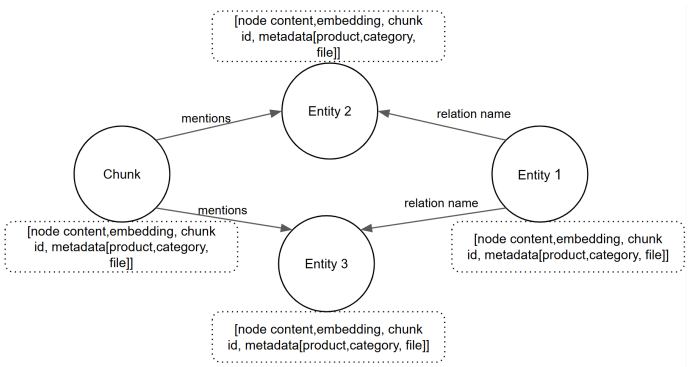


Figure 7: Snapshot of the constructed Knowledge Graph generated using LlamaIndex

Figure 7 presents a snapshot of the KG constructed using the LlamaIndex based pipeline. While the overall preprocessing workflow remains consistent across all three extraction methods—document chunking, metadata enrichment, embedding computation, and LLM-based extraction—the underlying modelling assumptions differ substantially. The Neo4j *SimpleKGPipeline* induces an explicit schema prior to extraction, the LangChain-based approach exposes individual entities and relations in a triple-oriented manner, and the *SimpleLLMPathExtractor* emphasizes the

recovery of longer relational paths.

Across all three methods, the general pattern is the same: the document collection is split into chunks, embeddings are computed for RAG-style retrieval, and an LLM-based extractor is invoked to identify entities and relations that are then written into Neo4j. The main differences lie in how the schema is induced, how explicitly the intermediate graph representation is exposed, and how tightly the extraction is tied to a specific RAG framework (Neo4j GraphRAG, LangChain, or LlamaIndex). Figure 8 compares the resulting graphs in terms of node and edge counts across all extraction methods. Notably, the SimpleLLMPathExtractor based on LlamaIndex yields a substantially higher number of entity nodes and relations compared to the other approaches. In addition, the number of distinct relation types is markedly higher for this method, indicating a much more expressive but also less constrained emergent schema.

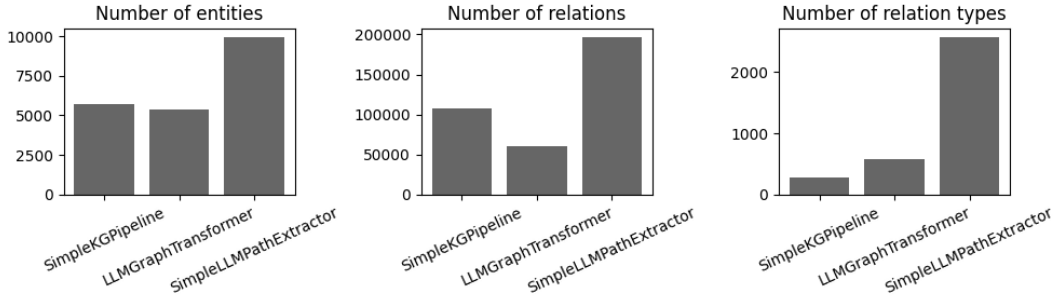


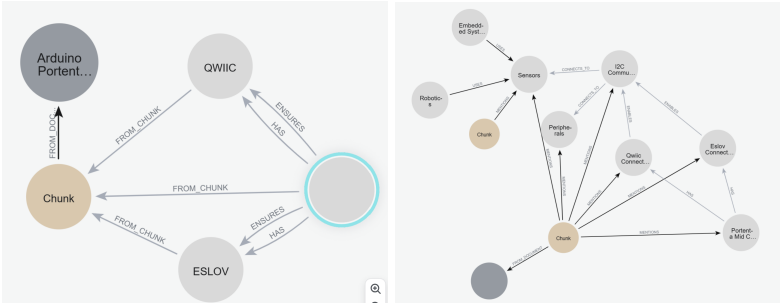
Figure 8: Structural Statistics of the Constructed Knowledge Graphs

To illustrate the differences in how each extraction pipeline constructs KGs, consider the following example chunk with the identifier *Arduino Portenta Mid Carrier Proto Shield::c10*. Despite its short length, this chunk contains multiple hardware components and functional relationships:

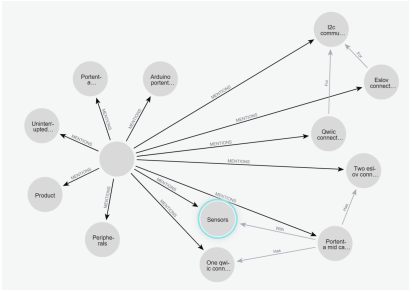
> *"The Portenta Mid Carrier Proto Shield has two ESLOV connectors and one QWIIC connector for I2C communication, ensuring uninterrupted connectivity with sensors and peripherals."*

The *SimpleKGPipeline* included in Neo4j GraphRAG performs lightweight entity and relation extraction optimized for speed and structural consistency. In this example, the pipeline detected three entities: the product *Portenta Mid Carrier Proto Shield*, two connector types: *ESLOV Connector* and *Qwiic Connector*. The resulting graph is minimal but clean, consisting primarily of *MENTIONS* relationships from the chunk node to each detected entity. No cross-chunk reasoning or inference takes place, and the pipeline does not attempt to expand or generalize entity classes. This behavior reflects the design goal of SimpleKG: provide a deterministic, low-noise representation that remains close to the original text.

In contrast, the *LLMGraphTransformer* produces a substantially richer graph structure. By prompting the LLM to generate semantically meaningful nodes and relationships, the transformer identifies not only the explicitly mentioned connectors but also higher-level concepts such as *Sensors*, *Peripherals*, and *I2C Communication*. Furthermore, it automatically links these concepts to related nodes originating from other chunks (e.g., previously extracted sensor concepts), enabling cross-document semantic consolidation. The generated relationships (e.g., *CONNECTS_TO*, *ENABLES*, *HAS*) are more diverse and reflect inferred functional dependencies.



(a) SimpleKGPipeline (b) LLMGraphTransformer



(c) SimpleLLMPathExtractor

Figure 9: Comparison of Knowledge Graph Extraction Methods on the same Chunk

The *LlamaIndex SimpleLLMPathExtractor* represents a third extraction paradigm, focusing on identifying path-structured knowledge rather than isolated triples. When applied to the same chunk, the extractor generates an even larger set of nodes and relationships. For instance, it produces multiple representations of connector entities, such as: *two ESLOV connectors* and *ESLOV connector* , one *QWIIC connector* and *QWIIC connector*. This reflects the extractor's tendency to preserve surface-level linguistic distinctions, resulting in fine-grained but semantically overlapping entities. Compared to the LLMGraphTransformer, the relationship naming is more conservative and often restricted to generic types such as *MENTIONS* or *HAS*. Nevertheless, the extractor successfully captures complex conceptual paths such as the role of connectors in enabling I2C-based communication.

3.4.4 Entity Resolution Methodology

In the literature, entity resolution in KGs is frequently addressed using graph embedding–based similarity methods, where entities are represented in a latent vector space and merged based on proximity measures [69]. Such approaches are effective when duplicate entities arise from semantic ambiguity and when the underlying graph structure is sufficiently dense and reliable.

In the present setting, however, duplicate entities predominantly result from heterogeneous naming conventions in technical documentation rather than from semantic ambiguity. Superficial syntactic variations such as punctuation, spacing, or capitalization often refer to the same real-world component, whereas small numeric differences encode semantically relevant distinctions and must be preserved. Since embedding-based similarity does not explicitly enforce numeric safety, it may lead to erroneous merges in this domain. For this reason, embedding-based entity resolution methods are not employed in this work. Instead, a hybrid approach combining deterministic, numeric-safe blocking with constrained LLM-based disambiguation is adopted.

We consider the problem of identifying and merging duplicate entities in a KG constructed automatically from technical documentation. Due to naming heterogeneity, multiple nodes may represent the same real-world entity, while small numeric differences must be treated as semantically meaningful. In this work, entity resolution is formulated as an automated duplicate-detection task that operates primarily on the id, name, and entityType attributes of entity nodes. The proposed approach therefore follows a two-stage pipeline consisting of candidate generation and semantic disambiguation.

Definition 6 (Entity Resolution Problem)**.** Let $\mathcal{E}$ denote the set of entity nodes in the KG. Each entity $e \in \mathcal{E}$ is associated with an identifier string $\mathrm{id}(e) \in \Sigma^*$ and a type label $\tau(e) \in \mathcal{T}$. The goal of entity resolution is to identify all entities that refer to the same real-world object and to consolidate them into a single graph node.

To efficiently identify potential duplicates, deterministic blocking strategies are applied to the entity identifiers. These strategies normalize identifier strings using numeric-safe transformations, plural handling, and token-based representations with generic-word filtering. Entities sharing the same blocking key and type are grouped as candidate sets, while groups of size one are discarded.

Definition 7 (Blocking Function)**.** A blocking function is a mapping

$$k : \mathcal{E} \to \Sigma^*$$

that assigns each entity to a normalization key derived from its identifier string. Blocking functions are designed to collapse superficial syntactic variations while preserving numeric meaning.

Each candidate set is subsequently processed by an LLM, which determines whether the contained identifiers refer to the same real-world entity. The disambiguation step explicitly allows syntactic variation but prevents merges that would conflate numerically distinct components. To ensure robustness, overlapping merge decisions are resolved using a union–find structure before the final merge is executed in the graph database.

The resulting merge operations consolidate duplicate nodes while preserving graph structure and relationships, yielding a more consistent and compact KG.

3.4.5 Building Community Summaries

As a further step in the refinement of the constructed KG, community structures are introduced to capture higher-level semantic groupings of entities. This refinement step organizes entities beyond individual relations by identifying clusters of semantically related nodes. To this end, the Leiden community detection method, which was introduced in Chapter 2, is applied to a similarity graph constructed over entity embeddings. The following section describes the construction of this similarity graph and the subsequent identification of hierarchical communities using graph-based clustering methods.

The procedure operates on an existing KG in which entity embeddings have already been computed and stored.

Let $\mathcal{E}$ denote the set of entity nodes in the graph. Each entity $e \in \mathcal{E}$ is associated with a vector representation $\mathbf{v}(e) \in \mathbb{R}^d$, which encodes semantic information derived from textual descriptions and identifiers extracted from technical documentation.

To capture semantic proximity between entities, an undirected similarity graph $G_{\text{sim}} = (\mathcal{E}, \mathcal{S})$ is constructed. An edge $(e_i, e_j) \in \mathcal{S}$ exists if and only if e_j belongs to the set of the k nearest neighbors of e_i in embedding space and the corresponding similarity exceeds a predefined threshold. Similarity between two entities e_i and e_j is computed using cosine similarity, which is defined as

$$\text{sim}(e_i, e_j) = \frac{\mathbf{v}(e_i) \cdot \mathbf{v}(e_j)}{\|\mathbf{v}(e_i)\| \, \|\mathbf{v}(e_j)\|}.$$

For each entity, only the top-k most similar neighbors with $\text{sim}(e_i, e_j) \geq \theta$ are retained, where θ denotes a fixed similarity cutoff. Each retained similarity relation is materialized as an undirected edge weighted by the corresponding similarity score.

In technical documentation such as Arduino manuals, this step connects entities whose textual representations exhibit strong semantic overlap, for instance references to closely related hardware components or functionally similar modules described using different terminology.

To enable scalable computation, entity nodes and their embedding properties are projected into an in-memory graph representation. The k-nearest-neighbor procedure is executed on this projection, and the resulting similarity relations are written

back to the persistent graph as explicit edges. This separation allows the computationally intensive similarity estimation to be performed efficiently while preserving the resulting structure for subsequent analysis.

Based on the constructed similarity graph, hierarchical community detection is performed to identify groups of semantically related entities. Let G_{sim} denote the weighted, undirected graph induced by similarity relations. The objective is to partition the entity set into communities such that entities within the same community exhibit high internal similarity, while similarity across communities is comparatively low.

For this purpose, the Leiden algorithm is applied to G_{sim}. The algorithm optimizes a modularity-based objective function and yields a hierarchy of community assignments. As a result, each entity $e \in \mathcal{E}$ is associated with an ordered sequence of community identifiers

$$\text{comm}(e) = (c_0, c_1, \ldots, c_L),$$

where c_0 denotes the finest-grained community and c_L denotes the coarsest-level community in the hierarchy.

The detected community structure is subsequently materialized in the KG by introducing explicit community nodes. For each community identifier occurring at a given hierarchical level, a corresponding community node is created. Each entity is linked to its associated community nodes via membership relations. To preserve the hierarchical structure, directed parent–child relations are introduced between community nodes of adjacent levels. This construction yields a directed acyclic community hierarchy shared across all entities and enables community-aware traversal and aggregation operations in downstream processing steps.

This step augments the previously induced community hierarchy with textual and semantic descriptors derived from the underlying technical documentation. For a fixed hierarchy level $\ell \in \mathbb{N}$, we consider the set of community nodes

$$\mathcal{C}_\ell = \{c \mid c \text{ is a community node with } \text{level}(c) = \ell\}.$$

For each community $c \in \mathcal{C}_\ell$, the procedure aggregates evidence from (i) member entities, (ii) associated product metadata, and (iii) representative documentation passages. The resulting aggregate is stored as a structured community profile and subsequently summarized by a language model.

Let $\mathcal{E}(c)$ denote the set of entity nodes assigned to c via membership relations. From $\mathcal{E}(c)$, we derive three primary collections. First, we extract a bounded list of entity identifiers

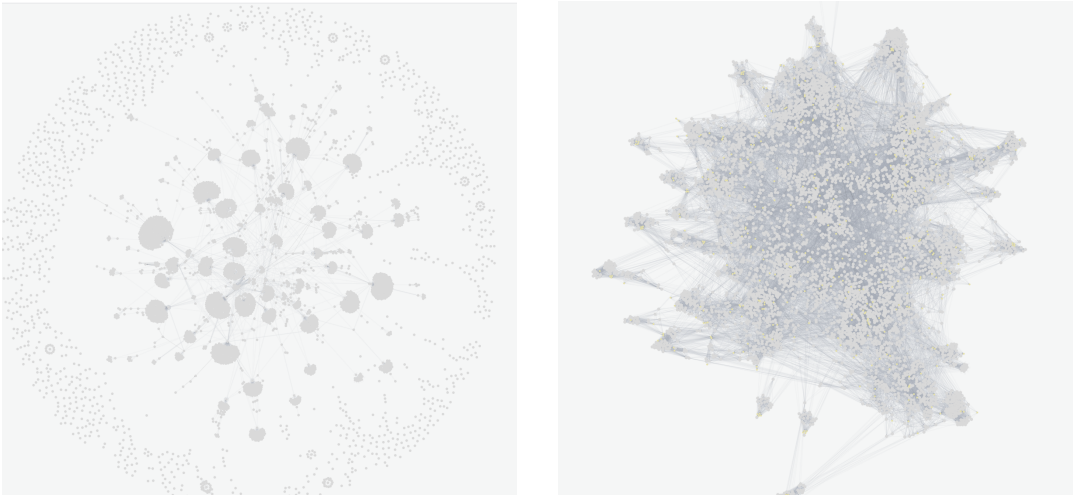
$$\text{Ent}(c) = \text{Top}_m(\{\text{id}(e) \mid e \in \mathcal{E}(c),\ \text{id}(e) \neq \perp\}),$$

where $\text{Top}_m(\cdot)$ returns the first m available values under a fixed ordering and $\perp$ denotes a missing attribute. Second, we form multisets of product and category

attributes,

$$\mathrm{Prod}(c) = [\,\mathrm{product}(e) \mid e \in \mathcal{E}(c),\ \mathrm{product}(e) \neq \bot\,],$$
$$\mathrm{Cat}(c) = [\,\mathrm{product\_category}(e) \mid e \in \mathcal{E}(c),\ \mathrm{product\_category}(e) \neq \bot\,].$$

and compute frequency maps to obtain the most prevalent items. In practice, this yields a compact characterization of which products and product families dominate a community at level ℓ. Third, the procedure retrieves representative text snippets from documentation chunks. Let $\mathcal{X}$ denote the set of chunk nodes, each carrying a text field $\mathrm{text}(x)$ and auxiliary metadata such as file name and product context. A chunk $x \in \mathcal{X}$ is considered evidence for community c if it mentions at least one entity in $\mathcal{E}(c)$. The resulting set of evidence snippets is then truncated to a fixed number q to control prompt length and computational cost. In an Arduino documentation setting, such snippets may include descriptions of board features, pin functions, sensor specifications, or library usage patterns, thereby providing community-specific context beyond isolated entity names.



(a) KG before community detection

(b) KG after community detection

Figure 10: KG created by SimpleKGPipeline before and after Community Detection

The aggregated information is serialized into a canonical JSON representation and stored as a community property full_content. Based on the same information, a language model is prompted to produce three outputs: a short topic label, a concise multi-sentence summary, and a list of key terms. The prompt constrains the model to produce strictly valid JSON with keys topic_label, summary, and key_terms, enabling deterministic parsing and direct persistence in the graph.

Formally, we may view the language-model component as a mapping

$$g : \mathcal{J} \to \mathcal{O},$$

where $\mathcal{J}$ denotes the space of JSON-serializable community profiles and $\mathcal{O}$ denotes the space of structured outputs containing a label, summary, and term list. For each community $c \in \mathcal{C}_\ell$, the pipeline computes an input profile $j(c) \in \mathcal{J}$, invokes $g(j(c))$, and writes the resulting fields back to the corresponding community node. If parsing fails, the implementation falls back to storing the raw model output as the summary, ensuring that the workflow remains robust under occasional format deviations. Figure 10 shows the KG created by SimpleKGPipeline before and after the application of the Leiden community detection algorithm. The left side depicts the graph based solely on embedding-derived similarity relations, while the right side illustrates the same graph after entities have been assigned to communities and organized accordingly.

3.5 Building Knowledge Graph-enhanced Retrievals

This subsection presents retrieval approaches that explicitly incorporate KG structures to enrich the retrieval process. It introduces KG-enhanced RAG retrieval strategies implemented with Neo4j GraphRAG, LlamaIndex Property Graphs libraries, and community-based representations, providing the methodological foundation for the subsequent feasibility study.

3.5.1 Knowlege Graph-enhanced Retrieval with Neo4j GraphRAG

This subsection introduces KG–enhanced retrieval strategies implemented using GraphRAG library in Neo4j. The methods are evaluated on two KG databases constructed using distinct pipelines: Neo4j's native *SimpleKGPipeline* and the *LLMGraphTransformer* provided by LangChain. The construction of these graphs was described in Sections 3.4.1 and 3.4.2, respectively.

The evaluated approaches extend dense and hybrid vector-based retrieval with explicit graph structures and relational information. All retrieval variants are implemented using the Neo4j GraphRAG libraries. Specifically, the following strategies are considered: *Dense + KG*, *Hybrid + KG*, *Hybrid + KG with Reranking*, and *Community KG*.

Each method is discussed individually in the following sections with respect to its retrieval mechanism.

Knowledge Graph-enhanced Dense Chunk Retrieval on the SimpleKGPipeline Database. The pipeline implements a RAG procedure that combines dense vector retrieval over `Chunk` nodes with local knowledge-graph expansion over `__Entity__` nodes. Given a user query, the retriever first identifies semantically similar seed chunks via embedding-based nearest-neighbour search. The retrieved seed chunks are then enriched with graph-derived evidence by (i) collecting entities directly linked to each seed chunk, (ii) retrieving additional chunks connected to those entities, and (iii) collecting related entities and multi-hop relations in the

entity subgraph. Finally, the generator conditions an LLM on the concatenated evidence (chunk texts and selected relation traces) to produce a context-grounded answer that is restricted to the retrieved information.

Let the property graph be denoted by $G = (V, E)$ with node types $\mathcal{T}_V = \{\texttt{Chunk}, \texttt{\_\_Entity\_\_}\}$ and relation types including `:FROM_CHUNK` and arbitrary entity–entity relations. Each chunk $c \in V_{\texttt{Chunk}}$ has text content $t(c) \in \Sigma^*$ and a pre-computed embedding vector $z(c) \in \mathbb{R}^d$. Let $e(\cdot) : \Sigma^* \to \mathbb{R}^d$ be the query embedding function.

Dense seed retrieval. For a user query $q \in \Sigma^*$, compute the query embedding

$$z_q = e(q) \in \mathbb{R}^d.$$

Define a similarity function (e.g., cosine similarity) $\mathrm{sim} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. The dense retriever selects the top-k seed chunks

$$S_k(q) = \mathrm{TopK}_{c \in V_{\texttt{Chunk}}} \ \mathrm{sim}(z_q, z(c)),$$

and returns a ranked list of pairs $(c, s(c))$ with score $s(c) = \mathrm{sim}(z_q, z(c))$. Here, k is a configurable hyperparameter controlling the number of seed chunks retrieved (set via `top_k` in the implementation).

Entity attachment to seed chunks. For each seed chunk $c \in S_k(q)$, the pipeline retrieves up to 20 directly linked entities via incoming `:FROM_CHUNK` edges:

$$E_1(c) = \{\, e \in V_{\texttt{\_\_Entity\_\_}} \mid (e \xrightarrow{\texttt{:FROM\_CHUNK}} c) \in E \,\}, \qquad E_1^{(20)}(c) = \mathrm{Trunc}_{20}(E_1(c)).$$

Intuitively, $E_1(c)$ are the entities mentioned in c.

Related chunk expansion via shared entities. Using the entity set $E_1^{(20)}(c)$, the retriever expands the context with additional chunks that are connected to these entities:

$$C_{\mathrm{rel}}(c) = \{\, c' \in V_{\texttt{Chunk}} \mid \exists e \in E_1^{(20)}(c) : (e \xrightarrow{\texttt{:FROM\_CHUNK}} c') \in E \,\}.$$

To prioritize chunks supported by many of the seed entities, define an evidence count for each candidate chunk

$$\mathrm{ev}(c') = \left| \{\, e \in E_1^{(20)}(c) \mid (e \xrightarrow{\texttt{:FROM\_CHUNK}} c') \in E \,\} \right|.$$

The pipeline keeps the top-10 related chunks by $\mathrm{ev}(c')$:

$$C_{\mathrm{top}}(c) = \mathrm{Top10}_{c' \in C_{\mathrm{rel}}(c)} \ \mathrm{ev}(c').$$

Related entities and multi-hop entity relations. The retriever also collects (i) entities adjacent to the seed entities (for logging/interpretability) and (ii) a set of entity–entity relations within 1 to 2 hops that explicitly excludes traversals using `:FROM_CHUNK`. Formally, define the one-hop neighbour entities

$$E_2(c) = \{\, e' \in V_{\texttt{\_\_Entity\_\_}} \mid \exists e \in E_1^{(20)}(c) : (e \leftrightarrow e') \in E \,\}, \qquad E_2^{(30)}(c) = \mathrm{Trunc}_{30}(E_2(c)).$$

For the relation set, let $\mathcal{R}$ be the set of relation types and exclude `:FROM_CHUNK`. Consider paths

$$p : e \xrightarrow{r_1} v_1 \xrightarrow{r_2} \cdots \xrightarrow{r_\ell} e', \qquad \ell \in \{1, 2\},$$

such that all intermediate nodes are entities ($v_i \in V_{\texttt{__Entity__}}$) and all traversed relation types satisfy $r_i \neq$ `:FROM_CHUNK`. The pipeline collects the set of traversed relationships:

$$\mathrm{Rels}(c) = \bigcup_{e \in E_1^{(20)}(c)} \bigcup_{\substack{p \in \mathcal{P}_{1..2}(e) \\ r_i(p) \neq \texttt{:FROM\_CHUNK}}} \{\, \text{relationships in } p \,\},$$

where $\mathcal{P}_{1..2}(e)$ denotes all entity-only paths of length 1 or 2 starting at e.

Context construction for generation. For each seed chunk c, define the chunk set used as textual evidence:

$$U(c) = \{c\} \cup C_{\mathrm{top}}(c).$$

The final context string is formed by concatenating the chunk texts and a textual rendering of the collected entity relations:

$$C(c) = \bigoplus_{x \in U(c)} t(x) \;\oplus\; \bigoplus_{r \in \mathrm{Rels}(c)} \mathrm{render}(r),$$

where $\oplus$ denotes concatenation and $\mathrm{render}(\cdot)$ maps a relationship $r = (u \xrightarrow{\tau} v)$ (with optional properties such as `details` or `description`) to a human-readable string (e.g., "$u - \tau \rightarrow v$").

(6) Context-grounded answer generation. Let LLM denote the answer model. The final answer is produced as

$$a = \mathrm{LLM}(q, C),$$

where C is the concatenation of the contexts associated with the retrieved results (in practice, the top-k retrieved items). The prompt constraints enforce that a should be grounded in C (i.e., the model is instructed to use only the provided context and to state explicitly when required information is missing).

Interpretation. The dense retrieval step selects candidate `Chunk` nodes by semantic similarity, while the graph expansion step enriches each seed with evidence anchored in `__Entity__` nodes and their local neighbourhood (related chunks via shared entities, and 1–2-hop entity relations excluding `:FROM_CHUNK`). The generator then performs conditional text generation given (q, C), yielding a RAG-style answer supported by both chunk text and explicit entity-relation traces.

Knowledge Graph-enhanced Hybrid Chunk Retrieval on the SimpleKGPipeline Database. This pipeline takes an identical overall approach to the *Dense +KG* pipeline, with the major difference being how the initial seed chunks are retrieved.

While the subsequent graph-based expansion and context construction steps remain unchanged, the seed selection relies on a hybrid retrieval strategy rather than purely dense vector similarity.

Let the property graph be $G = (V, E)$ with chunk nodes $V_{\texttt{Chunk}}$ and entity nodes $V_{\texttt{\_\_Entity\_\_}}$. Each chunk $c \in V_{\texttt{Chunk}}$ has text content $t(c) \in \Sigma^*$ and an embedding vector $z(c) \in \mathbb{R}^d$. For a user query $q \in \Sigma^*$, the corresponding query embedding is $z_q = e(q) \in \mathbb{R}^d$.

Hybrid seed retrieval. In contrast to dense retrieval, the hybrid retriever combines semantic similarity in the embedding space with lexical full-text matching. Let $\mathrm{sim}(z_q, z(c))$ denote a dense similarity function (e.g., cosine similarity) and let $\mathrm{lex}(q, t(c))$ denote a lexical relevance score obtained from a full-text index. Each chunk is assigned a hybrid score

$$s_{\mathrm{hyb}}(c \mid q) = \alpha \ \mathrm{sim}(z_q, z(c)) + (1 - \alpha) \ \mathrm{lex}(q, t(c)),$$

where $\alpha \in [0, 1]$ controls the relative contribution of semantic and lexical signals. The top-k seed chunks are then selected as

$$S_k(q) = \mathrm{TopK}_{c \in V_{\texttt{Chunk}}} \ s_{\mathrm{hyb}}(c \mid q),$$

with k configured via the `top_k` parameter in the implementation.

Graph-based expansion and generation. Starting from the selected seed chunks, the pipeline follows the same graph-based expansion strategy as in the dense variant, including entity attachment, related chunk selection based on shared entities, extraction of multi-hop entity relations, and context construction for answer generation. The final answer is produced by a language model conditioned exclusively on the constructed context.

Knowledge Graph-enhanced Hybrid Chunk Retrieval with Reranking on the SimpleKGPipeline Database. Let the property graph be denoted by $G = (V, E)$ with chunk nodes $V_{\texttt{Chunk}}$ and entity nodes $V_{\texttt{\_\_Entity\_\_}}$. Each chunk $c \in V_{\texttt{Chunk}}$ has text content $t(c) \in \Sigma^*$ and an embedding vector $z(c) \in \mathbb{R}^d$. For a user query $q \in \Sigma^*$, the corresponding embedding is $z_q = e(q)$.

Hybrid candidate retrieval. A hybrid retriever combines dense semantic similarity and lexical full-text matching to retrieve an initial candidate set. Let $\mathrm{sim}(z_q, z(c))$ denote a dense similarity score and $\mathrm{lex}(q, t(c))$ a lexical relevance score. Each chunk is assigned a hybrid score

$$s_{\mathrm{hyb}}(c \mid q) = \alpha \ \mathrm{sim}(z_q, z(c)) + (1 - \alpha) \ \mathrm{lex}(q, t(c)),$$

with $\alpha \in [0, 1]$. The retriever selects an enlarged candidate set

$$C_{k'}(q) = \mathrm{TopK'}_{c \in V_{\texttt{Chunk}}} \ s_{\mathrm{hyb}}(c \mid q), \qquad k' = \max(mk, k),$$

where k is the requested number of final results and m is a fixed multiplier.

Graph-based context construction. For each candidate chunk $c \in C_{k'}(q)$, a structured context representation is constructed. Let

$$E_1(c) = \{\, e \in V_{\texttt{\_\_Entity\_\_}} \mid (e \xrightarrow{\texttt{:FROM\_CHUNK}} c) \in E \,\}$$

denote the set of entities directly attached to c, truncated to at most 20 elements. Using $E_1(c)$, related chunks are collected via shared entities and ranked by evidence count, retaining the top 10 related chunks. In addition, up to 30 related entities and all entity–entity relations within one hop (excluding `:FROM_CHUNK`) are extracted.

The resulting context string for candidate c is defined as

$$\mathrm{context}(c) = \bigoplus_{x \in \{c\} \cup C_{\mathrm{rel}}(c)} t(x) \;\oplus\; \bigoplus_{r \in \mathcal{R}(c)} \mathrm{render}(r),$$

where $\oplus$ denotes concatenation and $\mathrm{render}(\cdot)$ maps a relation to a textual description.

Context-level reranking. Let $q \in \Sigma^*$ be a user query and let $C_{k'}(q) = \{c_1, \ldots, c_{k'}\}$ denote the candidate set produced by the hybrid retriever, where $k' > k$. Each candidate c_i is associated with a constructed context representation $\mathrm{context}(c_i)$. A cross-encoder re-ranker assigns a relevance score to each query–context pair:

$$s_{\mathrm{re}}(c_i \mid q) = \mathrm{CE}\big(q, \mathrm{context}(c_i)\big),$$

where CE jointly encodes the query and the context and directly estimates their semantic compatibility. The final result set is obtained by sorting candidates according to s_{re} and selecting the top-k contexts:

$$S_k(q) = \mathrm{TopK}_{c_i \in C_{k'}(q)}\; s_{\mathrm{re}}(c_i \mid q).$$

(4) Context-grounded answer generation. The final answer is generated by a language model conditioned exclusively on the concatenation of the re-ranked context representations:

$$a = \mathrm{LLM}\left(q, \bigoplus_{c \in S_k(q)} \mathrm{context}(c)\right),$$

with prompt constraints enforcing that the response is grounded in the provided context and explicitly signals missing information when required.

Graph-based context and generation. For each selected seed chunk $c \in S_k(q)$, the context string $\mathrm{context}(c)$ is constructed using the same graph operations as in the non-reranked variants: (i) attach up to 20 entities via incoming `:FROM_CHUNK`, (ii) select up to 10 related chunks by shared-entity evidence, (iii) collect up to 30 related entities, and (iv) extract entity–entity relations within one hop while excluding `:FROM_CHUNK`. The final answer is generated by the LLM conditioned only on the concatenation of the retrieved contexts.

Knowledge Graph-enhanced Retrieval on the LLMGraphTransformer Database. The retrieval pipelines applied to the KGs constructed using the LLMGraphTransformer pipeline follow the same overall methodological structure as those defined for the SimpleKG-based database. Accordingly, the approaches *Knowledge Graph-enhanced Hybrid Chunk Retrieval*, *Knowledge Graph-enhanced Dense Chunk Retrieval*, and *Knowledge Graph-enhanced Hybrid Chunk Retrieval with Reranking* are also employed on this database.

The primary difference lies in the underlying graph schema, which differs from the SimpleKGPipeline database schema due to the construction strategy of the LLMGraphTransformer pipeline. This schema variation, introduced and discussed in Chapter 3, affects the concrete graph structure but does not alter the fundamental retrieval logic. To avoid redundancy, the detailed descriptions of the individual retrieval pipelines are therefore not repeated here; instead, the reader is referred to the corresponding sections describing the three retrieval strategies above.

3.5.2 Knowlege Graph-enhanced Retrieval with LlamaIndex Property Graph

This section introduces KG-enhanced retrieval strategies implemented using the LlamaIndex Property Graph framework on database constructed with SimpleLLMPathExtractor. The methods are evaluated on KG databases constructed using LlamaIndex-based ingestion pipelines, which store chunk and entity nodes together with their relations in a property graph backed by Neo4j. The construction of these property graphs was described in the preceding subsections.

Knowledge Graph-enhanced Dense Chunk Retrieval on the SimpleLLMPathExtractor This approach implements a KG-enhanced retrieval strategy using the LlamaIndex Property Graph retriaval libraries. The retrieval pipeline follows a two-stage design.

In the first stage, a dense vector retriever is used to identify a small set of semantically relevant *seed chunks*. Each chunk stored in the property graph is associated with an embedding vector, and semantic similarity between the user query and chunk embeddings is used to select promising entry points into the graph. This dense retrieval step serves exclusively as a mechanism for seed selection and does not yet incorporate any graph structure.

In the second stage, each retrieved seed chunk is expanded through structured traversal of the underlying KG in Neo4j. Starting from a seed chunk node, the method retrieves directly mentioned entity nodes, additional chunks connected via shared entities, and relations between entities within a bounded hop distance. This expansion step enriches the retrieved context with relational and cross-document evidence that cannot be captured by vector similarity alone. The expanded subgraph is serialized into a unified textual context, which is subsequently provided to the language model for answer generation.

Formal Description Let q denote a natural language question, let $\mathcal{X}$ be the set of

chunk nodes, $\mathcal{E}$ the set of entity nodes, and $\mathcal{R}$ the set of relations in the property graph.

Dense Seed Retrieval Each chunk $x \in \mathcal{X}$ is associated with an embedding vector $\mathbf{v}_x \in \mathbb{R}^d$, and the query q is embedded as $\mathbf{v}_q$. Using cosine similarity, the top-k most relevant seed chunks are selected as

$$R_k(q) = \underset{x \in \mathcal{X}}{\overset{k}{\arg\operatorname{top}}} \cos(\mathbf{v}_q, \mathbf{v}_x)$$

The resulting set $R_k(q)$ defines the entry points for graph-based expansion.

Entity Expansion For each seed chunk $x \in R_k(q)$, the set of directly mentioned entities is defined as

$$E(x) = \{\, e \in \mathcal{E} \mid (x, e) \in \text{MENTIONS} \,\}.$$

Related Chunk Expansion To incorporate additional contextual evidence, chunks that share entities with the seed chunk are retrieved:

$$C_{\text{rel}}(x) = \{\, x' \in \mathcal{X} \setminus \{x\} \mid \exists e \in E(x) : (x', e) \in \text{MENTIONS} \,\}.$$

These chunks are ranked by the number of shared entities and truncated to a fixed maximum.

Entity–Entity Relation Expansion To capture structured relational knowledge, relations between entities reachable within a bounded hop distance are extracted:

$$\mathcal{R}(x) = \{\, r \in \mathcal{R} \mid \exists e_1, e_2 \in \mathcal{E} : e_1 \in E(x),\ (e_1, e_2) \in r,\ \text{hop}(e_1, e_2) \leq 2 \,\}.$$

excluding relations that directly involve chunk nodes.

Graph-Augmented Context Construction The graph-augmented context for a seed chunk x is constructed as

$$\mathcal{C}(x) = \text{concat}\Big(\textsf{text}(x), \{\textsf{text}(x') \mid x' \in C_{\text{rel}}(x)\}, \{\textsf{serialize}(r) \mid r \in \mathcal{R}(x)\}\Big).$$

Aggregating over all seed chunks yields the final context

$$\mathcal{C}(q) = \bigcup_{x \in R_k(q)} \mathcal{C}(x).$$

Answer Generation Finally, the language model generates an answer

$$a = \text{LLM}(q, \mathcal{C}(q)),$$

subject to the constraint that only information contained in the retrieved graph-augmented context $\mathcal{C}(q)$ may be used.

Knowledge Graph-enhanced Hybrid Chunk Retrieval with Reranker on SimpleLLMPathExtractor database. This retrieval approach combines dense vector similarity with symbolic signals derived from the property graph. Candidate chunks are retrieved using a hybrid strategy that integrates embedding-based retrieval with graph-aware keyword and synonym expansion. It then extends the hybrid GraphRAG retrieval pipeline by introducing an explicit reranking stage. An initial hybrid retriever first produces an enlarged candidate set of context items, which are subsequently re-scored using a cross-encoder reranking model conditioned on the user query. Only the top-ranked contexts are retained and enriched through KG expansion. The final re-ranked and graph-augmented context is then used for downstream answer generation.

3.5.3 Community Retrievers

This subsection introduces retrieval strategies that leverage community-level representations of the KG. These approaches operate on aggregated graph structures to explore how higher-level abstractions influence retrieval behavior and downstream answer generation.

Community Retriever Only. This retrieval strategy operates exclusively on precomputed `__Community__` nodes (level 1) that represent aggregated summaries of the KG and was implemented on LlamaIndex Property Graph. Given a user question q, the pipeline first applies *HyDE* (Hypothetical Document Embeddings) by generating a hypothetical community-style summary $\tilde{c}$ that would likely contain an answer to q. This synthetic text is embedded into a dense vector and used to query a Neo4j vector index over communities. The retrieved communities are then *expanded* by collecting associated `__Entity__` nodes and referenced `Chunk` nodes via the schema $(c : \_\_Community\_\_)$–[: `IN_COMMUNITY`]–$(e : \_\_Entity\_\_)$–[: `MENTIONS`]–$(ch$: `Chunk`$)$. Finally, the answer is generated by an LLM constrained to the assembled community-centric context.

Formally, let q denote a question. HyDE produces a hypothetical community report

$$\tilde{c} = g_{\text{HyDE}}(q),$$

where $g_{\text{HyDE}}(\cdot)$ is a language model conditioned on q and a community-report template. Next, an embedding model $f(\cdot)$ maps text to a dense vector space $\mathbb{R}^d$:

$$\mathbf{v} = f(\tilde{c}) \in \mathbb{R}^d.$$

Let $\mathcal{C}_1$ be the set of all level-1 community nodes, each with a stored embedding vector $\mathbf{u}(c) \in \mathbb{R}^d$. Community retrieval is performed by selecting the top-k communities according to a vector similarity function $\mathrm{sim}(\cdot,\cdot)$ (e.g., cosine similarity):

$$R_k(q) = \mathrm{TopK}_{c \in \mathcal{C}_1} \ \mathrm{sim}\big(\mathbf{v}, \mathbf{u}(c)\big).$$

For each retrieved community $c \in R_k(q)$, the method expands graph context by collecting (i) a bounded set of community entities

$$E(c) = \{\, e \in \mathcal{E} \mid (c, e) \in \text{IN\_COMMUNITY} \,\} \quad \text{(truncated to at most } M_E \text{ entities).}$$

and (ii) a bounded set of chunks mentioned by those entities

$$Ch(c) = \{\, ch \in \mathcal{CH} \mid \exists e \in E(c) : (e, ch) \in \text{MENTIONS} \,\} \quad \text{(truncated to at most } M_{ch} \text{ chunks).}$$

Here, $\mathcal{E}$ denotes the set of `__Entity__` nodes and $\mathcal{CH}$ the set of `Chunk` nodes. The final context $X(q)$ provided to the answer generator is the concatenation of the retrieved community texts and the sampled entity/chunk texts:

$$X(q) = \bigoplus_{c \in R_k(q)} \Big(c.\texttt{full\_content} \oplus \bigoplus_{e \in E(c)} \phi(e) \oplus \bigoplus_{ch \in Ch(c)} \psi(ch) \Big),$$

where $\oplus$ denotes concatenation, $\phi(e)$ is a short entity representation (e.g., name/identifier), and $\psi(ch)$ is the textual content of a chunk. Finally, an answer is produced by an LLM constrained to use only $X(q)$:

$$a = g_{\text{ans}}(q, X(q)).$$

This realizes a community-centric dense retrieval approach in which similarity search is performed at the abstraction level of communities, while graph structure is used only for controlled expansion into `__Entity__` and `Chunk` evidence.

Hybrid Community Retrieval. This pipeline combines dense vector retrieval and sparse lexical retrieval over `Chunk` nodes and subsequently enriches the retrieved evidence with community-level summaries. The approach is implemented on a databases constructed using the *LLMGraphTransformer* and *SimpleKGPipeline*.

Given a question q, the *HybridCypherRetriever* first retrieves a candidate set of text chunks by jointly applying (i) a dense similarity search over chunk embeddings and (ii) a sparse full-text search over a lexical index. The union of both result sets is then re-ranked using a cross-encoder, which scores the semantic relevance between the question and each chunk text more precisely than the initial retrieval stage.

For each selected chunk, the pipeline expands the context by retrieving directly mentioned neighboring nodes, treated as `__Entity__` nodes, as well as the associated level-1 `__Community__` nodes connected via the relation `:IN_COMMUNITY`. Answer generation is performed by a GraphRAG component that conditions an LLM on the assembled chunk evidence and the related community summaries, while explicitly restricting the model to the provided context.

Formally, let $\mathcal{X}$ denote the set of chunk nodes $x \in \mathcal{X}$, each with text $\text{text}(x)$ and an embedding vector $\mathbf{u}(x) \in \mathbb{R}^d$. The hybrid retrieval stage computes two relevance signals for a query q: (i) a dense score based on the embedding $\mathbf{v}(q) = f(q)$,

$$s_{\text{dense}}(q, x) = \text{sim}\big(\mathbf{v}(q), \mathbf{u}(x)\big),$$

and (ii) a sparse score based on full-text matching,

$$s_{\text{sparse}}(q, x) = \text{FT}(q, \text{text}(x)),$$

where $\text{FT}(\cdot, \cdot)$ denotes the fulltext retrieval score (e.g., BM25-style scoring). The hybrid retriever returns a candidate pool

$$\mathcal{K}(q) = \text{Cand}\big(\{(x, s_{\text{dense}}(q, x))\}_{x \in \mathcal{X}}, \{(x, s_{\text{sparse}}(q, x))\}_{x \in \mathcal{X}}\big),$$

where $\text{Cand}(\cdot)$ denotes the internal fusion strategy of the hybrid retriever that combines dense and sparse results into a unified candidate list.

Next, a cross-encoder reranker $r(\cdot, \cdot)$ assigns a more faithful relevance estimate to each candidate chunk by directly encoding the pair $(q, \text{text}(x))$:

$$s_{\text{rerank}}(q, x) = r\big(q, \text{text}(x)\big).$$

The final top-k chunks are selected by

$$R_k(q) = \text{TopK}_{x \in \mathcal{K}(q)} \; s_{\text{rerank}}(q, x).$$

To integrate community information, the method expands each retrieved chunk $x \in R_k(q)$ using the graph structure. Let $E(x)$ be the set of directly mentioned non-chunk neighbors of x (in the code via `:MENTIONS` and the constraint $\neg$`Chunk`):

$$E(x) = \{\, e \mid (x)\texttt{MENTIONS}(e) \;\wedge\; e \notin \mathcal{X} \,\}.$$

Let $\mathcal{C}_1$ be the set of level-1 communities, and let $C(x)$ denote the communities reachable from x through entities:

$$C(x) = \{\, c \in \mathcal{C}_1 \mid \exists e \in E(x) : (e)\texttt{IN\_COMMUNITY}(c) \,\}.$$

The final context supplied to the answer generator is the concatenation of chunk texts and associated community summaries:

$$X(q) = \bigoplus_{x \in R_k(q)} \Big(\text{text}(x) \;\oplus\; \bigoplus_{c \in C(x)} c.\texttt{full\_content}\Big),$$

(where $\oplus$ denotes concatenation and c.`full_content` is the textual summary stored on the community node). Finally, the generated answer is produced by an LLM conditioned on q and $X(q)$:

$$a = g_{\text{ans}}(q, X(q)).$$

Thus, the method is *hybrid* at retrieval time (dense + sparse over chunks) and *community-enriched* at context construction time (injecting level-1 `__Community__` summaries linked to retrieved evidence).

3.5.4 Language Models and Embedding Models

All retrieval-augmented generation (RAG) and KG-enhanced RAG pipelines in this work are based on OpenAI-provided models for both text generation and vector embedding. For answer generation, the GPT-5 model is employed via the Neo4j GraphRAG interface, with the specific model configuration controlled through environment variables to ensure consistent usage across experiments. For vector-based retrieval, semantic representations of text chunks and queries are computed using the `text-embedding-3-small` embedding model. These embeddings are used uniformly across all dense and hybrid retrieval pipelines, enabling a controlled comparison between baseline RAG and graph-augmented retrieval approaches.

4 Feasibility Study

This chapter investigates the practical feasibility of integrating KGs into retrieval-augmented generation pipelines by means of a structured, exploratory assessment. The focus lies on analysing observed effects across different retrieval configurations rather than conducting a statistically conclusive evaluation.

4.1 Study Design

This section describes the experimental setup used to compare baseline and graph-augmented RAG pipelines under controlled conditions. It outlines the construction of the question set, the categorization of query types, and the rationale behind the chosen comparison framework. In particular, it details the composition of the question set and the definition of the considered question types, which form the basis for all subsequent analyses.

To enable a systematic comparison between the baseline RAG system and the graph-augmented RAG approach, a curated gold-standard question set comprising 160 questions was constructed. This dataset covers five distinct query types, each representing a different cognitive or structural challenge for retrieval-based systems. By evaluating both pipelines across the same typed questions, we can analyse where graph-structured information provides measurable advantages.

Factual Queries. Factual queries request a single explicit piece of information that is explicitly stated in the documentation. They do not require inference or the integration of multiple sources.Factual questions primarily depend on retrieving an accurate chunk, although KGs may improve precision where explicit attributes are stored:

- *How much memory does the Arduino Due provide?*
- *What is the maximum sampling rate of the BMP390 pressure sensor?*

Reasoning Queries. Reasoning queries require deriving a conclusion that is not explicitly stated in any single passage. This may include causal reasoning, comparative reasoning, or determining suitability based on constraints:

- *Why is the Arduino Due more suitable for complex real-time applications than the Mega 2560 Rev3?*
- *I tried to program my Nano RP2040 Connect, but nothing works and the board is not recognized correctly. I used a different Arduino IDE version than usual. Could this be the reason, and what should I do next?*

Disambiguation Queries. Disambiguation queries require determining which of several possible entities a question refers to. These queries test whether the retrieval system can use contextual or structural signals to resolve ambiguity:

- *Is the FCC compliance information referring to the Arduino Due or Mega 2560 Rev3?*
- *I see that both Nano 33 IoT and Nano RP2040 Connect support Wi-Fi, but are they equivalent in terms of computing capabilities?*

Multi-hop Queries. Multi-hop queries require combining multiple relational steps to reach the correct answer. This often involves reasoning across several connected entities:

- *Which board contains a total of three different motion-related sensors?*
- *I powered the Portenta Machine Control and connected it via USB, but I still cannot start programming. What steps do I need to complete before coding?*

Summary Queries. Summary queries require aggregating several pieces of information into a synthesized answer rather than retrieving one fact.These questions test whether the system can integrate information scattered across multiple chunks or graph nodes:

- *Which processor family / platform is used by the largest number of Arduino products?*
- *Which Arduino products integrate or embed another Arduino product as part of their design?*

4.2 Assessment Method

For the assessment of generated answers, this work adopts an *LLM-as-a-Judge* assessment paradigm following the methodology proposed by [33]. The judge model evaluates responses with respect to *Answer Relevance*, *Completeness*, *Correctness*, and *Helpfulness*, where the criterion for *Answer Relevance* is inspired by the RAG evaluation framework of [27].

The assessment is conducted across the three document databases introduced in Section 3.4 and compares multiple RAG architectures. In addition to four RAG baseline variants, a set of knowledge-graph-enhanced and graph-augmented RAG methods is assessed. The evaluated methods are defined and described in Sections 3.3 and 3.6.

The four RAG baselines comprise a naive dense vector retriever as well as three advanced variants based on sparse retrieval, dense retrieval, and a hybrid sparse–dense retriever. The graph-based and KG-enhanced methods include hybrid retrievers operating on entity-centric KGs and property graphs, optionally combined with community-aware retrieval and neural reranking. All methods are assessed using an identical evaluation protocol and a shared set of assessment metrics, which are defined in the following.

Answer Relevance. The answer relevance metric estimates whether a generated answer a is semantically aligned with the user question q, independent of any external ground truth. The implemented approach follows a *reverse-QA* idea: a judge LLM generates a small set of questions that the answer could plausibly address, and semantic similarity between these generated questions and the original question is used as the relevance signal. This reduces reliance on lexical overlap and allows relevance to be measured even when paraphrasing occurs. In the implementation, semantic similarity is computed in an embedding space using cosine similarity.

Formally, let $q \in \mathcal{T}$ be the user question and $a \in \mathcal{T}$ the system answer, where $\mathcal{T}$ denotes the space of texts. A judge LLM produces m questions $Q(a) = \{\tilde{q}_1, \ldots, \tilde{q}_m\}$ that could be answered by a. Let $e(\cdot) : \mathcal{T} \to \mathbb{R}^d$ be the embedding function and let

$$\cos(u, v) = \frac{u^\top v}{\|u\|_2 \, \|v\|_2}$$

denote cosine similarity. The continuous relevance score is

$$s_{\text{rel}}(q, a) = \frac{1}{m} \sum_{i=1}^{m} \cos\big(e(q), e(\tilde{q}_i)\big),$$

where m is the number of generated questions and d is the embedding dimension. The implementation maps the internal score to a 1–5 rating via rounding:

$$\text{Rel}_{1\text{–}5}(q, a) = \text{round}\big(1 + 4 \cdot \text{clip}_{[0,1]}(s_{\text{rel}}(q, a))\big),$$

where $\text{clip}_{[0,1]}(x) = \min(1, \max(0, x))$ ensures a valid range.

Completeness. The completeness metric quantifies how fully an answer covers the expected solution content, expressed as a set of gold steps/items. The judge LLM decomposes both the gold answer and the model answer into comparable step-level units and estimates omissions and incorrect inclusions at the level of these

units. This produces a scalar completeness score that captures *coverage of required content*, rather than stylistic quality. The implementation supports both single-list (steps only) and multi-list (steps and causes) variants; here the scoring principle is identical and can be stated for a single expected set.

Formally, let $G = \{g_1, \ldots, g_n\}$ be the set of n expected steps/items (gold), and let $A = \{a_1, \ldots, a_m\}$ be the set of m steps/items implied by the produced answer, where both sets are extracted by the judge LLM under semantic equivalence. Let $M \subseteq G$ be the subset of missing gold items and let $W \subseteq A$ be the subset of wrong (non-equivalent) items. The judge outputs omission and error rates

$$r_{\text{miss}} = \frac{|M|}{|G|}, \qquad r_{\text{wrong}} = \frac{|W|}{|G|},$$

and an overall completeness score $s_{\text{comp}} \in [0, 1]$ (in the code this is provided directly by the judge LLM). The reported 1–5 score is obtained by

$$\text{Comp}_{1\text{–}5}(q, a, G) = \text{round}\big(1 + 4 \cdot s_{\text{comp}}\big),$$

where q denotes the question context and G the gold steps/items.

Correctness and Recall. The correctness metric evaluates whether a generated answer conveys the *core factual content* specified by a gold reference, while explicitly penalizing answers that introduce *severe* misleading or contradictory claims. In contrast to completeness, which emphasizes the presence of all procedural steps and their ordering, correctness focuses on *semantic coverage* of the gold items and a coarse-grained assessment of error severity. The implementation operationalizes correctness via a judge-estimated coverage score and an auxiliary severity label. In addition, answers that constitute meta-responses (e.g., refusals, missing-information statements, or non-answers) are treated as false negatives. For quantitative reporting in this thesis, only the coverage-derived 1–5 score is used.

Formally, let $G = \{g_1, \ldots, g_n\}$ denote the set of key gold facts/items required to answer question q, and let a be the generated answer. A judge model estimates the coverage

$$c(q, a, G) \in [0, 1],$$

interpreted as the fraction of gold items that are correctly present in a (up to semantic equivalence). For reporting, the implementation maps coverage to a discrete 1–5 score by

$$\text{Corr}_{1\text{–}5}(q, a, G) = \text{round}\big(1 + 4 \cdot c(q, a, G)\big).$$

Here, q is included because the assessment of whether an item g_i is addressed depends on the question context, and G provides the reference set of required content.

Besides the numeric score, the code assigns a categorical outcome

$$y \in \{\text{TP}, \text{PARTIAL}, \text{FP}, \text{FN}\},$$

using two coverage thresholds τ_{TP} and τ_{P} and an error severity label

$$s \in \{\text{low}, \text{medium}, \text{high}\}.$$

In the relaxed configuration used in the experiments, $\tau_{\mathrm{TP}} = 0.60$ and $\tau_{\mathrm{P}} = 0.20$. Meta-responses are mapped to FN. Otherwise, the classification is defined by

$$y = \begin{cases} \mathrm{TP}, & c(q,a,G) \geq \tau_{\mathrm{TP}} \ \wedge \ s \neq \text{high}, \\ \mathrm{PARTIAL}, & \tau_{\mathrm{P}} \leq c(q,a,G) < \tau_{\mathrm{TP}} \ \wedge \ s \neq \text{high}, \\ \mathrm{FP}, & c(q,a,G) < \tau_{\mathrm{P}} \ \vee \ s = \text{high}, \\ \mathrm{FN}, & a \text{ is a meta-response.} \end{cases}$$

The severity variable s captures whether the answer contains major incorrect claims that would mislead a user; in particular, $s = \text{high}$ forces the outcome to FP even if coverage is non-trivial. While this categorical judgement is logged for diagnostic purposes, the quantitative correctness value reported in this thesis is $\mathrm{Corr}_{1\text{–}5}$ as defined above. **Recall.** Recall measures the fraction of questions for which a fully correct answer was produced. Let TP, FN, and $PARTIAL$ denote the numbers of true positives, false negatives, and partially correct answers, respectively. In the strict assessment setting used in this thesis, partially correct answers are treated as incorrect. Recall is therefore defined as

$$\text{Recall} = \frac{TP}{TP + FN + PARTIAL}.$$

Precision is analogously defined as the proportion of fully correct answers among all generated answers, given by

$$\text{Precision} = \frac{TP}{TP + FP + PARTIAL}.$$

Helpfulness. Helpfulness captures the practical utility of an answer for a technical support setting, emphasizing actionable steps, clarity, and directness. In the implementation, a judge LLM first assigns a raw helpfulness rating on a 1–5 scale based on a rubric (from "not helpful" to "highly helpful"). This raw utility score is then *faithfulness-gated*: answers that are not supported by context should not be rewarded as highly helpful, even if they appear plausible. Therefore, the final helpfulness rating combines the raw helpfulness with the faithfulness score using a multiplicative gate controlled by a base parameter. To compute the helpfulness score, a context-based faithfulness assessment is performed beforehand. Faithfulness measures whether the claims made in an answer can be supported by the retrieved context by decomposing the answer into a limited set of atomic statements and verifying each statement against the context using a judge LLM. The individual verdicts are aggregated into a normalized faithfulness score by assigning graded weights to supported, partially supported, and unsupported statements. This score reflects the

degree to which the answer is grounded in the provided evidence. The resulting faithfulness score is then used as a multiplicative gate when computing the final helpfulness rating, ensuring that answers lacking contextual support are not rated as highly helpful, even if they appear plausible.

Formally, let $\mathrm{Help}^{\mathrm{raw}}_{1\text{–}5}(q,a) \in \{1,2,3,4,5\}$ be the judge-provided raw helpfulness rating for question q and answer a. Convert it to $[0,1]$ via

$$h_{\mathrm{raw}} = \frac{\mathrm{Help}^{\mathrm{raw}}_{1\text{–}5}(q,a) - 1}{4}.$$

Let $s_{\mathrm{faith}}(a,C) \in [0,1]$ be the faithfulness score defined above and let $\beta \in [0,1]$ be the gating base (in the code: `HELP_GATE_BASE`). The gated helpfulness score is

$$h_{\mathrm{final}} = h_{\mathrm{raw}} \cdot \Big(\beta + (1-\beta)\, s_{\mathrm{faith}}(a,C)\Big),$$

and the reported final 1–5 value is

$$\mathrm{Help}^{\mathrm{final}}_{1\text{–}5}(q,a,C) = \mathrm{round}\big(1 + 4 \cdot \mathrm{clip}_{[0,1]}(h_{\mathrm{final}})\big).$$

Here, β ensures that the score does not collapse to zero when faithfulness is low, while still strongly penalizing unfaithful yet seemingly useful answers.

4.3 Findings

This section reports the empirical results obtained from the experimental assessment of the different retrieval pipelines. The findings summarize observed performance patterns across all assessment metrics without making causal claims.

4.3.1 Observed Impact on Answer Quality

This section provides a descriptive analysis of the observed performance patterns, highlighting differences in answer relevance, completeness, helpfulness, precision, and recall. Table 2 presents the aggregated results for all retrieval pipelines across the considered quality metrics. For brevity, the following abbreviations are used in the tables: *SimpleKG* for the SimpleKGPipeline database, *KG* for Knowledge Graph-enhanced, *LlamaIndex* for the SimpleLLMPathExtractor database, and *LLMGraph* for the LLMGraphTransformer database.

LLMGraph_Dense_KG achieves solid scores in answer relevance and helpfulness, while completeness is comparatively lower. Precision and recall fall within a moderate range, indicating a balanced but not optimal coverage of the evaluated questions.

LLMGraph_Hybrid_KG shows consistent performance across all assessment metrics. Both precision and recall are higher than in the dense configuration, while answer relevance and helpfulness remain at a comparable level.

LLMGraph_Hybrid_KG_Reranker attains stable values across the quality metrics and demonstrates balanced precision and recall scores. Compared to the non-reranked hybrid variant, the observed differences remain moderate.

LLMGraph_Community_Hybrid achieves the highest completeness score among the LLMGraph-based retrieval pipelines. Precision and recall remain in the mid-range, while the qualitative metrics indicate stable overall performance.

LlamaIndex_Dense_KG exhibits good answer relevance but lower completeness and recall values. Precision remains moderate, suggesting limited coverage despite acceptable answer accuracy.

LlamaIndex_Hybrid_KG_Rerank reaches high helpfulness and completeness scores. Precision and recall are situated in the upper mid-range, indicating an improved balance between answer quality and coverage.

LlamaIndex_Community_Only consistently shows lower precision and recall values. The qualitative metrics are also lower compared to hybrid and knowledge-graph-based configurations.

RAG_Advanced_Dense records substantially lower scores across all assessment metrics. Both precision and recall are particularly low, indicating limited effectiveness for this retrieval pipeline.

RAG_Advanced_Hybrid achieves high helpfulness scores while maintaining moderate precision and recall. The quality-related metrics are comparatively strong, whereas coverage remains uneven.

RAG_Advanced_Sparse demonstrates high values in answer relevance, completeness, and helpfulness. Precision and recall are comparatively high, indicating balanced performance across assessment dimensions.

RAG_Naive_Hybrid exhibits consistently low values across all metrics. In particular, precision and recall are limited, suggesting weak overall performance.

SimpleKG_Dense_KG achieves stable quality scores with moderate precision and recall. The results indicate consistent but not outstanding performance.

SimpleKG_Hybrid_KG shows improved precision and recall compared to the dense SimpleKG variant. The qualitative metrics remain at a high and stable level.

SimpleKG_Hybrid_KG_Rerank attains the highest completeness score among the SimpleKG configurations. Both precision and recall are among the highest observed values, indicating strong overall performance.

SimpleKG_Community_Hybrid achieves high precision and recall alongside stable qualitative scores. Overall, this retrieval pipeline ranks among the strongest configurations reported in the table.

Overall, the results reveal clear performance differences across the evaluated retrieval pipelines. Hybrid and knowledge-graph-based configurations generally achieve higher precision and recall than dense or naive approaches, while community-based and reranking extensions are often associated with improved quality metrics.

Table 2: Results across Retrieval Pipelines

Retrieval Pipeline	Answer Relevance	Completeness	Helpfulness	Precision	Recall
LLMGraph_Dense_KG	3.71	2.74	4.51	0.717	0.647
LLMGraph_Hybrid_KG	3.79	2.99	4.31	0.792	0.797
LLMGraph_Hybrid_KG_Reranker	3.72	2.96	4.38	0.764	0.743
LLMGraph_Community_Hybrid	3.70	3.02	4.22	0.740	0.720
LlamaIndex_Dense_KG	3.70	2.46	4.17	0.703	0.557
LlamaIndex_Hybrid_KG_Rerank	3.71	3.06	4.59	0.733	0.723
LlamaIndex_Community_Only	3.60	2.52	3.85	0.592	0.514
RAG_Advanced_Dense	3.42	1.42	2.00	0.180	0.065
RAG_Advanced_Hybrid	3.75	2.88	4.56	0.660	0.683
RAG_Advanced_Sparse	3.78	2.95	4.63	0.671	0.689
RAG_Naive_Hybrid	2.40	1.39	2.26	0.286	0.111
SimpleKG_Dense_KG	3.71	2.99	4.22	0.737	0.701
SimpleKG_Hybrid_KG	3.76	2.83	4.28	0.776	0.750
SimpleKG_Hybrid_KG_Rerank	3.79	3.12	4.27	0.789	0.768
SimpleKG_Community_Hybrid	3.77	2.88	4.19	0.832	0.788

4.3.2 Usefulness by Query Type

This subsection analyzes the assessment metrics across different query types, examining how performance varies depending on the nature of the query. The results are compared across retrieval pipelines to highlight observed differences in usefulness-related metrics for each query category.

Table 3 summarizes recall scores for each retrieval script, grouped by query type.

Disambiguation questions show consistently higher recall values for hybrid and knowledge-graph-based retrieval pipelines. In particular, *SimpleKG_Hybrid_KG* and *LlamaIndex_Hybrid_KG_Rerank* achieve the highest recall scores in this category, whereas dense-only and naive RAG pipelines exhibit substantially lower recall.

Factual questions are handled most effectively by advanced hybrid and sparse RAG configurations. *RAG_Advanced_Hybrid* and *RAG_Advanced_Sparse* achieve the highest recall values, while community-only and dense LlamaIndex pipelines show comparatively weaker performance.

Multi-hop questions highlight clear differences across retrieval pipelines. Hybrid SimpleKG-based pipelines, particularly *SimpleKG_Hybrid_KG_Rerank* and

SimpleKG_Hybrid_KG, achieve the highest recall scores, whereas dense and naive retrieval configurations demonstrate limited coverage for multi-hop queries.

Reasoning questions are answered with higher recall by hybrid and community-enhanced pipelines. *SimpleKG_Community_Hybrid* and *LLMGraph_Hybrid_KG* reach the strongest performance, while dense-only and naive RAG pipelines again exhibit lower recall values.

Summary questions show more variation across retrieval pipelines. Hybrid and reranked SimpleKGPipelines achieve the highest recall, whereas dense and naive RAG configurations record substantially lower values.

Overall, the results indicate that hybrid retrieval pipelines consistently achieve higher recall across most query types compared to dense-only and naive approaches. Knowledge-graph-based and reranking-enhanced configurations generally exhibit stronger coverage for complex query types such as multi-hop and reasoning questions, while dense and naive pipelines show limited recall across all categories.

Tables 4–6 report answer relevance, completeness, and helpfulness scores by query type for all retrieval pipelines.

Disambiguation questions exhibit generally mid to high answer relevance scores across most retrieval pipelines, with *SimpleKG_Community_Hybrid* and *SimpleKG_Hybrid_KG* achieving the highest values. Completeness scores for this query type remain mostly in the mid range, where community-enhanced and hybrid pipelines tend to perform better than dense-only approaches. Helpfulness is consistently high across knowledge-graph-based pipelines, with *RAG_Advanced_Hybrid* and *LlamaIndex_Hybrid_KG_Rerank* reaching the highest scores.

Factual questions show high answer relevance for hybrid and sparse retrieval pipelines, particularly *SimpleKG_Hybrid_KG_Rerank* and *RAG_Advanced_Sparse*. Completeness varies more strongly across pipelines, with hybrid and community-based configurations achieving higher values than dense-only variants. Helpfulness scores are highest for advanced RAG and hybrid pipelines, while dense and naive approaches consistently score lower.

Multi-hop questions reveal moderate answer relevance across most retrieval pipelines, with hybrid and reranking-enhanced SimpleKGPipelines achieving slightly higher values. Completeness scores are generally lower than for factual queries, although hybrid pipelines such as *SimpleKG_Hybrid_KG_Rerank* and *LlamaIndex_Hybrid_KG_Rerank* exhibit comparatively stronger performance. Helpfulness is consistently high for hybrid and knowledge-graph-based pipelines, whereas dense and naive pipelines show reduced usefulness for multi-hop queries.

Reasoning questions are associated with moderate answer relevance across all pipelines, with community-enhanced configurations achieving marginally higher values. Completeness scores remain in the mid range, with hybrid pipelines outperforming dense-only and naive approaches. Helpfulness is highest for hybrid and advanced RAG pipelines, while dense configurations demonstrate slightly lower scores.

Table 3: Recall Scores by Query Type across Retrieval Pipelines

Retrieval Pipeline	Disambiguation	Factual	Multi-hop	Reasoning	Summary
LLMGraph_Dense_KG	0.741	0.633	0.621	0.733	0.541
LLMGraph_Hybrid_KG	0.875	0.714	0.828	0.857	0.744
LLMGraph_Hybrid_KG_Reranker	0.808	0.643	0.793	0.793	0.694
LLMGraph_Community_Hybrid	0.731	0.690	0.724	0.724	0.730
LlamaIndex_Dense_KG	0.654	0.433	0.679	0.571	0.486
LlamaIndex_Hybrid_KG_Rerank	0.880	0.679	0.767	0.750	0.595
LlamaIndex_Community_Only	0.455	0.393	0.467	0.680	0.564
RAG_Advanced_Dense	0.045	0.033	0.069	0.077	0.094
RAG_Advanced_Hybrid	0.846	0.900	0.571	0.769	0.400
RAG_Advanced_Sparse	0.793	0.867	0.621	0.679	0.500
RAG_Naive_Hybrid	0.154	0.033	0.241	0.125	0.029
SimpleKG_Dense_KG	0.714	0.655	0.828	0.741	0.581
SimpleKG_Hybrid_KG	0.885	0.690	0.900	0.786	0.543
SimpleKG_Hybrid_KG_Rerank	0.692	0.679	0.933	0.733	0.784
SimpleKG_Community_Hybrid	0.840	0.633	0.893	0.893	0.725

Summary questions show relatively stable answer relevance across most retrieval pipelines, with hybrid and community-based approaches achieving the strongest values. Completeness scores for this query type remain moderate overall, with reranking-enhanced SimpleKGPipelines achieving the highest scores. Helpfulness is consistently high across most hybrid pipelines, whereas naive and dense RAG configurations yield notably lower usefulness scores.

Table 4: Answer Relevance Scores by Query Type across Retrieval Pipelines

Retrieval Pipeline	Disambiguation	Factual	Multi-hop	Reasoning	Summary
LLMGraph_Dense_KG	3.63	3.87	3.67	3.57	3.80
LLMGraph_Hybrid_KG	3.73	4.13	3.67	3.57	3.85
LLMGraph_Hybrid_KG_Reranker	3.80	4.10	3.60	3.53	3.62
LLMGraph_Community_Hybrid	3.60	3.87	3.53	3.73	3.75
LlamaIndex_Dense_KG	3.83	3.83	3.73	3.43	3.68
LlamaIndex_Hybrid_KG_Rerank	3.63	3.97	3.73	3.50	3.72
LlamaIndex_Community_Only	3.63	3.53	3.57	3.37	3.82
RAG_Advanced_Dense	3.33	3.37	3.43	3.43	3.50
RAG_Advanced_Hybrid	3.80	4.00	3.57	3.77	3.65
RAG_Advanced_Sparse	3.73	4.03	3.53	3.80	3.78
RAG_Naive_Hybrid	2.53	1.67	2.97	2.60	2.28
SimpleKG_Dense_KG	3.77	3.80	3.67	3.67	3.68
SimpleKG_Hybrid_KG	3.87	4.07	3.70	3.43	3.75
SimpleKG_Hybrid_KG_Rerank	3.83	4.17	3.63	3.60	3.75
SimpleKG_Community_Hybrid	3.90	4.03	3.57	3.57	3.78

Table 5: Completeness Scores by Query Type across Retrieval Pipelines

Retrieval Pipeline	Disambiguation	Factual	Multi-hop	Reasoning	Summary
LLMGraph_Dense_KG	2.53	3.13	2.90	2.60	2.60
LLMGraph_Hybrid_KG	2.80	3.50	2.77	3.00	2.92
LLMGraph_Hybrid_KG_Reranker	2.93	3.47	2.87	2.97	2.65
LLMGraph_Community_Hybrid	3.43	3.57	2.77	2.90	2.60
LlamaIndex_Dense_KG	2.60	2.60	2.43	2.23	2.45
LlamaIndex_Hybrid_KG_Rerank	3.03	3.37	3.30	2.90	2.78
LlamaIndex_Community_Only	2.67	2.53	2.43	2.47	2.50
RAG_Advanced_Dense	1.30	1.13	1.60	1.37	1.65
RAG_Advanced_Hybrid	3.23	3.23	3.07	2.50	2.50
RAG_Advanced_Sparse	3.07	3.60	2.97	2.50	2.70
RAG_Naive_Hybrid	1.57	1.03	1.60	1.37	1.40
SimpleKG_Dense_KG	3.13	3.40	3.17	2.93	2.50
SimpleKG_Hybrid_KG	3.00	3.20	3.17	2.60	2.35
SimpleKG_Hybrid_KG_Rerank	2.97	3.53	3.27	3.07	2.85
SimpleKG_Community_Hybrid	2.90	3.30	3.07	2.70	2.52

Table 6: Helpfulness Scores by Query Type across Retrieval Pipelines

Retrieval Pipeline	Disambiguation	Factual	Multi-hop	Reasoning	Summary
LLMGraph_Dense_KG	4.53	3.87	4.73	4.77	4.62
LLMGraph_Hybrid_KG	4.27	4.33	4.27	4.27	4.40
LLMGraph_Hybrid_KG_Reranker	4.47	4.03	4.67	4.37	4.38
LLMGraph_Community_Hybrid	4.23	4.20	4.33	4.20	4.15
LlamaIndex_Dense_KG	4.47	3.37	4.53	4.27	4.20
LlamaIndex_Hybrid_KG_Rerank	4.77	4.03	4.90	4.60	4.65
LlamaIndex_Community_Only	3.80	3.33	4.00	3.97	4.08
RAG_Advanced_Dense	2.17	1.17	2.13	2.27	2.20
RAG_Advanced_Hybrid	4.80	4.67	4.40	4.53	4.45
RAG_Advanced_Sparse	4.70	4.87	4.53	4.33	4.70
RAG_Naive_Hybrid	2.57	1.23	3.00	2.57	2.00
SimpleKG_Dense_KG	4.33	4.07	4.33	4.23	4.15
SimpleKG_Hybrid_KG	4.57	4.20	4.37	4.10	4.20
SimpleKG_Hybrid_KG_Rerank	4.23	4.20	4.40	4.17	4.32
SimpleKG_Community_Hybrid	4.30	4.10	4.17	4.20	4.18

5 Extended Discussion and Conclusions

The experimental results of this thesis highlight several recurring patterns that provide deeper insight into the behavior of both standard RAG pipelines and KG-enhanced retrieval architectures in the context of technical documentation. In particular, the findings indicate that retrieval strategies relying exclusively on dense semantic embeddings consistently underperformed compared to hybrid approaches that combine dense and sparse retrieval signals—both in pure RAG configurations and in KG-enhanced variants.

A plausible explanation for this observation lies in the nature of technical documentation itself. The evaluated corpus, derived from Arduino product documentation, contains a high density of domain-specific terminology, component names, interface standards, and hardware identifiers. Many of these terms exhibit strong semantic similarity at the embedding level while referring to distinct physical components or functional roles. Dense retrieval alone may conflate such concepts due to semantic proximity, whereas sparse lexical signals preserve exact token matches that are crucial for disambiguation in technical contexts.

Hybrid retrieval pipelines benefit from combining these complementary signals. While dense retrieval captures broader semantic relevance, sparse retrieval ensures that exact component names, identifiers, and protocol references are retained. This synergy appears particularly important for queries that require precise identification of components or interfaces, which is common in hardware-related technical support scenarios. The consistently stronger performance of hybrid pipelines across multiple assessment metrics supports this interpretation.

A second notable finding concerns the differential impact of KG-enhanced retrieval across query types. The results show that hybrid RAG pipelines without KGs performed particularly well on factual queries. These queries often require retrieving a single, well-defined piece of information—such as a specification value, supported interface, or compatibility statement—that is explicitly stated within a document chunk. In such cases, the additional abstraction introduced by graph-based representations does not necessarily provide an advantage and may even introduce noise or dilute the most relevant evidence.

This effect is especially visible when comparing hybrid RAG pipelines to community-based KG variants on factual questions. Community-only pipelines, which rely exclusively on summarized community nodes, consistently performed worse in this query category. These summary nodes aggregate information across multiple entities and documents, which can obscure fine-grained factual details that are essential for answering narrowly scoped technical questions. In contrast, hybrid KG pipelines that retained access to both original chunks and community representations mitigated this limitation, achieving more balanced performance across query types.

The comparison between community-only and community-hybrid variants further underscores the importance of granularity in KG-enhanced retrieval. Community-only configurations intentionally restrict retrieval to higher-level summary nodes in order to reduce redundancy and improve abstraction. However, the results suggest that this abstraction alone is insufficient to support robust question answering in technical domains. Without access to the original textual evidence contained in chunks or entity descriptions, the generated context may lack the specificity required for accurate and complete answers. Hybrid configurations that combine community summaries with chunk- and entity-level context appear to strike a more effective balance between abstraction and detail.

Beyond retrieval quality, the experiments revealed a significant practical challenge related to context size. KG-enhanced retrievers frequently produced substantially larger contexts than chunk-based RAG pipelines. Graph traversal, neighborhood expansion, and community aggregation often introduced multiple related entities, relations, and summaries into the retrieved context. While this enriched context can improve coverage and recall, it also increases the risk of exceeding model context limits or approaching them closely. In several configurations, the retrieved context was considerably larger than the original document chunks alone, highlighting an inherent trade-off between contextual richness and computational efficiency.

From a practical perspective, the manual construction of such KGs would have introduced substantial overhead and would not have been feasible at scale for the considered document collection. In contrast, the LLM-based extraction approaches demonstrated that large language models are capable of identifying a wide range of relevant entities and relations directly from unstructured technical text with comparatively low engineering effort.

Across the evaluated extraction pipelines, LLMs proved particularly effective at recognizing domain-specific relationships that would otherwise require extensive manual schema design or rule-based extraction. Even in short and dense documentation fragments, the models were able to infer meaningful semantic links, suggesting that LLM-driven graph construction provides a viable and scalable alternative to handcrafted knowledge engineering in technical domains. At the same time, the observed differences between extraction methods highlight an important trade-off between structural consistency and semantic richness, which has direct implications for downstream retrieval and reasoning tasks.

Taken together, these findings suggest that KGs should not be viewed as a universally superior replacement for standard RAG retrieval mechanisms. Instead, their effectiveness depends on careful integration with existing retrieval strategies and a clear understanding of the target query types. Dense-only retrieval appears insufficient for technical domains where lexical precision matters, while overly abstract graph-based retrieval can obscure critical details. Hybrid approaches—both at the retrieval level (dense + sparse) and at the representation level (chunks + entities + communities)—consistently provided the most robust performance across metrics.

In conclusion, this thesis demonstrates that KG-enhanced RAG systems offer meaningful benefits for domain-specific technical question answering when designed with attention to granularity, retrieval diversity, and context management. The results emphasize that structured knowledge is most effective when it complements, rather than replaces, unstructured textual evidence. These insights contribute to a more nuanced understanding of how RAG and KGs can be combined effectively and provide guidance for the design of practical hybrid architectures in technical and industrial applications.

References

[1] Abdelrahman Abdallah, Jamshid Mozafari, Bhawna Piryani, Mohammed Ali, and Adam Jatowt. From retrieval to generation: Comparing different approaches. *arXiv preprint arXiv:2502.20245*, February 2025.

[2] Abdelrahman Abdallah, Bhawna Piryani, Jamshid Mozafari, Mohammed Ali, and Adam Jatowt. How good are LLM-based rerankers? an empirical analysis of state-of-the-art reranking models. *arXiv preprint arXiv:2508.16757*, August 2025.

[3] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020. Retrieved from arXiv:2001.09977.

[4] Mehwish Alam, Davide Buscaldi, Michael Cochez, Francesco Osborne, Diego Reforgiato Recupero, Harald Sack, Özge Sevgili, Artem Shelmanov, Mikhail Arkhipov, Alexander Panchenko, and Chris Biemann. Neural entity linking: A survey of models based on deep learning. *Semantic Web*, 13(3):527–570, 2022.

[5] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *arXiv preprint arXiv:1610.06264*, 2017. Version 3.

[6] Blaise Agüera y Arcás. Do large language models understand us? *Daedalus*, 151(2):183–197, 2022.

[7] Arduino. Arduino documentation, 2025. Accessed: December 9, 2025.

[8] Angels Balaguer, Vinarma Benara, Renato Cunha, Roberto Estevão, Todd Hendry, Daniel Holstein, Jennifer Marsman, Nick Mecklenburg, Sara Malvar, Leonardo O. Nunes, Rafael Padilha, Morris Sharp, Bruno Silva, Swati Sharma, Vijay Aski, and Ranveer Chandra. Rag vs fine-tuning: Pipelines, tradeoffs, and a case study on agriculture. *arXiv preprint arXiv:2401.08406*, 2024.

[9] Michael K. Bergman. A common sense view of knowledge graphs, 2019. Adaptive Information, Adaptive Innovation, Adaptive Infrastructure Blog.

[10] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008, 2008.

[11] Piero Andrea Bonatti, Stefan Decker, Axel Polleres, and Valentina Presutti. Knowledge graphs: New directions for knowledge representation on the semantic web. *Dagstuhl Reports*, 8(9):29–111, 2018.

[12] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 2020.

[13] Anton Chernyavskiy, Dmitry Ilvovsky, and Preslav Nakov. Transformers: "the end of history" for nlp? *arXiv preprint arXiv:2105.00813*, 2021. Submitted on 9 Apr 2021, last revised 23 Sep 2021.

[14] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

[15] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *CoRR*, abs/2204.02311, 2022.

[16] Richard Cyganiak, David Wood, and Markus Lanthaler. Rdf 1.1 concepts and abstract syntax. W3c recommendation, World Wide Web Consortium (W3C), February 2014.

[17] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pages 16344–16359, 2022.

[18] Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 933–941. PMLR, 2017.

[19] Kartik Detroja, C. K. Bhensdadia, and Brijesh S. Bhatt. A survey on relation extraction. *Intelligent Systems with Applications*, 19:200244, 2023.

[20] Docling. Docling. Accessed: 2026-01-03.

[21] Alex Johannes Albertus Donkers, Dujuan Yang, and Nico Baken. Linked data for smart homes: Comparing rdf and labeled property graphs. In *Proceedings of the 8th Linked Data in Architecture and Construction Workshop (LDAC2020)*, Online, 2020. Eindhoven University of Technology. Accessed: December 9, 2025.

[22] Martin Dürst and Michel Suignard. Internationalized resource identifiers (iris). Rfc 3987, Internet Engineering Task Force (IETF), 2005.

[23] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha Metropolitansky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graph rag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, 2024.

[24] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, Dasha MetropolitanSky, Robert Osazuwa Ness, and Jonathan Larson. From local to global: A graphrag approach to query-focused summarization. *arXiv preprint arXiv:2404.16130*, February 2025.

[25] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In *Joint Proceedings of the Posters and Demos Track of the 12th International Conference on Semantic Systems (SEMANTiCS 2016) and the 1st International Workshop on Semantic Change & Evolving Semantics (SuCCESS'16)*, volume 1695 of *CEUR Workshop Proceedings*, page 4. CEUR-WS, 2016.

[26] Lisa Ehrlinger and Wolfram Wöß. Towards a definition of knowledge graphs. In *Proceedings of the SEMANTiCS 2016*, 2016.

[27] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval-augmented generation. *arXiv preprint arXiv:2309.15217*, 2025. Exploding Gradients; CardiffNLP, Cardiff University.

[28] Santo Fortunato. Community detection in graphs. *arXiv preprint arXiv:0906.0612*, 2010. Originally submitted June 2009, revised January 2010.

[29] Bin Fu, Yunqi Qiu, Chengguang Tang, Yang Li, Haiyang Yu, and Jian Sun. A survey on complex question answering over knowledge base: Recent advances and challenges. *arXiv preprint arXiv:2007.13069*, 2020.

[30] Philip Gage. A new algorithm for data compression. *C Users Journal*, 12(2):23–38, February 1994.

[31] Luyu Gao, Xinyu Ma, Jimmy Lin, and Jamie Callan. Precise zero-shot dense retrieval without relevance labels. *arXiv preprint arXiv:2212.10496*, 2022.

[32] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, March 2024.

[33] Jiawei Gu, Xuhui Jiang, Zhichao Shi, Hexiang Tan, Xuehao Zhai, Chengjin Xu, Wei Li, Yinghan Shen, Shengjie Ma, Honghao Liu, Saizhuo Wang, Kun Zhang, Zhouchi Lin, Bowen Zhang, Lionel Ni, Wen Gao, Yuanzhuo Wang, and Jian Guo. A survey on llm-as-a-judge. *arXiv preprint arXiv:2510.12345*, 2025. Accessed: 2025-12-06.

[34] Bernal Jimenez Gutierrez, Yiheng Shu, Yu Gu, Michihiro Yasunaga, and Yu Su. Hipporag: Neurobiologically inspired long-term memory for large language models. In *Advances in Neural Information Processing Systems*, volume 37, pages 59532–59569, 2024.

[35] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. REALM: Retrieval-augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning*, Proceedings of Machine Learning Research. JMLR.org, 2020.

[36] Haoyu Han, Li Ma, Yu Wang, Harry Shomer, Yongjia Lei, Kai Guo, Zhigang Hua, Bo Long, Hui Liu, Charu C. Aggarwal, and Jiliang Tang. Rag vs. graphrag: A systematic evaluation and key insights. *arXiv preprint arXiv:2502.11371*, 2025.

[37] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A. Rossi, Subhabrata Mukherjee, Xianfeng Tang, Qi He, Zhigang Hua, Bo Long, Tong Zhao, Neil Shah, Amin Javari, Yonglong Xia, and Jiliang Tang. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309*, 2025. Version 2.

[38] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2023. Version 5.

[39] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutierrez, José Emilio Labra Gayo, Sabrina Kirrane, Sebastian Neumaier, Axel Polleres, Roberto Navigli, Axel-Cyrille Ngonga Ngomo, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Computing Surveys*, 2021.

[40] Yizheng Huang and Jimmy X. Huang. The survey of retrieval-augmented text generation in large language models. *arXiv preprint arXiv:2404.10981*, 2024. Version 2.

[41] IBM. Ibm graphrag. Accessed: 2026-01-03.

[42] Sheng Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. A survey on knowledge graphs: Representation, acquisition, and applications. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[43] Ziwei Ji, Nayeon Lee, Rita Frieske, Tianyu Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Jang Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38, 2023.

[44] Di Jin, Binbin Zhang, Yue Song, Dongxiao He, Zhiyong Feng, Shizhan Chen, Weihao Li, and Katarzyna Musial. Modmrf: A modularity-based markov random field method for community detection. *Neurocomputing*, 405:218–228, 2020.

[45] Daniel Jurafsky and James H. Martin. Speech and language processing. `https://web.stanford.edu/~jurafsky/slp3/`, 2024. Third edition draft, pp. 4–198.

[46] Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. Large language models struggle to learn long-tail knowledge. *arXiv preprint arXiv:2211.08411*, 2023. Submitted on 15 Nov 2022, last revised 27 Jul 2023.

[47] 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)*, pages 6769–6781, Online, 2020. Association for Computational Linguistics.

[48] Mayank Kejriwal. Named entity resolution in personal knowledge graphs. *arXiv preprint arXiv:2307.12173*, 2023. Submitted on 22 Jul 2023.

[49] Michael Klesel and H. Felix Wittmann. Retrieval-augmented generation (rag). *Business & Information Systems Engineering*, 67(4):551–561, 2025.

[50] David Knoke and Song Yang. *Social Network Analysis*. SAGE Publications, 2019.

[51] Simon Knollmeyer, Oğuz Caymazer, and Daniel Grossmann. Document graphrag: Knowledge graph enhanced retrieval augmented generation for document question answering within the manufacturing domain. *Electronics*, 14(11):2102, 2025.

[52] David Krueger, Tegan Maharaj, Július Kramár, Mohammad Pezeshki, Nicolas Ballas, Nan Rosemary Ke, Anirudh Goyal, Yoshua Bengio, Aaron Courville, and Chris Pal. Zoneout: Regularizing rnns by randomly preserving hidden activations. *arXiv preprint arXiv:1606.01305*, 2016.

[53] Taku Kudo. Subword regularization: Improving neural network translation models with multiple subword candidates. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pages 66–75, 2018.

[54] John D. Lafferty and ChengXiang Zhai. Document language models, query models, and risk minimization for information retrieval. In *Proceedings of the 24th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 111–119. ACM, 2001.

[55] langchain. langchain. Accessed: 2026-01-03.

[56] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 9459–9474, 2020.

[57] llamaindex. llamaindex. Accessed: 2026-01-03.

[58] Xiaofei Ma, Yeyun Gong, Pengcheng He, Hao Zhao, and Nan Duan. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283*, 2023.

[59] Andrea Matarazzo and Riccardo Torlone. A survey on large language models with some insights on their capabilities and limitations. *arXiv preprint arXiv:2501.04040*, February 2025. Version 2.

[60] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. On faithfulness and factuality in abstractive summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1906–1919, Online, 2020. Association for Computational Linguistics.

[61] Microsoft. Microsoft graphrag. Accessed: 2026-01-03.

[62] Congmin Min, Sahil Bansal, Joyce Pan, Abbas Keshavarzi, Rhea Mathew, and Amar Viswanathan Kannan. Towards practical graphrag: Efficient knowledge graph construction and hybrid retrieval at scale. *arXiv preprint arXiv:2507.03226*, 2025.

[63] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models. *ACM Transactions on Intelligent Systems and Technology*, 16(5):1–72, 2025.

[64] Tapas Nayak, Navonil Majumder, Pawan Goyal, and Soujanya Poria. Deep neural approaches to relation triplets extraction: A comprehensive survey. *Cognitive Computation*, 13(5):1215–1232, 2021.

[65] neo4j. neo4jgraphrag. Accessed: 2026-01-03.

[66] neo4jgraphrag. neo4jgraphragpage. Accessed: 2026-01-03.

[67] Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. A review of relational machine learning for knowledge graphs. *Proceedings of the IEEE*, 104(1):11–33, 2016.

[68] Natasha Noy, Yuqing Gao, Anshu Jain, and Anant Narayanan. Industry-scale knowledge graphs: lessons and challenges. *Communications of the ACM*, 62(8):36–43, July 2019.

[69] Daniel Obraczka, Jonathan Schuchart, and Erhard Rahm. Eager: Embedding-assisted entity resolution for knowledge graphs. *arXiv preprint arXiv:2101.06126*, 2021. Submitted on 15 Jan 2021.

[70] Jeff Z. Pan, Giovanni Vetere, Jose Manuel Gomez-Perez, and Hai Wu. *Exploiting Linked Data and Knowledge Graphs in Large Organisations*. Springer International Publishing, Cham, Switzerland, 2017.

[71] Andrea Papaluca, Daniel Kref, Sergio Mendez Rodriguez, Artem Lensky, and Hanna Suominen. Zero- and few-shot knowledge graph triplet extraction with large language models. *arXiv preprint arXiv:2312.01954*, December 2023.

[72] Heiko Paulheim. Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8(3):489–508, 2017.

[73] Sachin Pawar, Pushpak Bhattacharyya, and Girish K. Palshikar. Techniques for jointly extracting entities and relations: A survey. *CoRR*, abs/2103.06118, 2021.

[74] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. Graph retrieval-augmented generation: A survey. *arXiv preprint arXiv:2408.08921*, 2024.

[75] Boci Peng, Yun Zhu, Yongchao Liu, Xiaohe Bo, Haizhou Shi, Chuntao Hong, Yan Zhang, and Siliang Tang. Graph retrieval-augmented generation: A survey. *arXiv preprint arXiv:2408.08921*, 2024.

[76] Wenqiang Peng, Guoliang Li, Yifan Jiang, Zhe Wang, Di Ou, Xinyu Zeng, and Enhong Chen. Large language model based long-tail query rewriting in taobao search. *arXiv preprint arXiv:2311.03758*, 2023.

[77] Ethan Perez, Siddharth Karamcheti, Rob Fergus, Jason Weston, Douwe Kiela, and Kyunghyun Cho. Finding generalizable evidence by learning to convince Q&A models. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Hong Kong, China, 2019. Association for Computational Linguistics.

[78] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, Alexander H. Miller, and Sebastian Riedel. Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*, 2019. Submitted on 3 Sep 2019, last revised 4 Sep 2019.

[79] Ofir Press, Noah A. Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Re*.

[80] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):1–9, 2019. Introduced GPT-2.

[81] Stephen E. Robertson and Steve Walker. On relevance weights with little relevance information. In *Proceedings of the 20th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 16–24. ACM, 1997.

[82] Stephen E. Robertson and Hugo Zaragoza. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends in Information Retrieval*, 3(4):333–389, 2009.

[83] Diego Sanmartin. Kg-rag: Bridging the gap between knowledge and creativity. *arXiv preprint arXiv:2405.12035*, 2024.

[84] Edward W. Schneider. Course modularization applied: The interface system and its implications for sequence control and data analysis. In *Proceedings of the Association for the Development of Instructional Systems (ADIS)*, Chicago, Illinois, 1973. Presented in April 1972.

[85] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, volume 1, pages 1715–1725, 2016.

[86] Stephan Seufert, Patrick Ernst, Srikanta J. Bedathur, Sarath Kumar Kondreddi, Klaus Berberich, and Gerhard Weikum. Instant espresso: Interactive analysis of relationships in knowledge graphs. In *Proceedings of the 25th International Conference on World Wide Web Companion*, pages 251–254, Republic and Canton of Geneva, Switzerland, 2016. International World Wide Web Conferences Steering Committee.

[87] Murray Shanahan. Talking about large language models. *arXiv preprint arxiv.org/abs/2212.03551*, 2022. Version 5.

[88] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[89] Amit Singhal. Introducing the knowledge graph: Things, not strings. `https://www.blog.google/products/search/introducing-knowledge-graph-things-not/`, 2012. Accessed: December 9, 2025.

[90] Xinying Song, Alex Salcianu, Yang Song, Dave Dopson, and Denny Zhou. Fast wordpiece tokenization. *arXiv preprint arXiv:2012.15524*, 2021. Version 3.

[91] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[92] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmad Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[93] Ross Taylor, Marcin Kardas, Guillem Cucurull, Thomas Scialom, Anthony Hartshorn, Elvis Saravia, Andrew Poulton, Viktor Kerkez, and Robert Stojnic. Galactica: A large language model for science. *CoRR*, abs/2211.09085, 2022.

[94] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *CoRR*, 2023.

[95] Vincent A. Traag, Ludo Waltman, and Nees Jan Van Eck. From louvain to leiden: Guaranteeing well-connected communities. *Scientific Reports*, 9(1):5233, 2019.

[96] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, 2017.

[97] Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Superglue: A stickier benchmark for general-purpose language understanding systems. In *Proceedings of the Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[98] Yuwei Wang, Nedim Lipka, Ryan A. Rossi, Albert Siu, Rongyao Zhang, and Tyler Derr. Knowledge graph prompting for multi-document question answering. *arXiv preprint arXiv:2305.18883*, 2023.

[99] Jonathan J. Webster and Chunyu Kit. Tokenization as the initial phase in NLP. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING)*, volume 4, 1992.

[100] Junde Wu, Jiayuan Zhu, Yunli Qi, Jingkun Chen, Min Xu, Filippo Menolascina, and Vicente Grau. Medical graph rag: Towards safe medical large language model via graph retrieval-augmented generation. *arXiv preprint arXiv:2408.04187*, 2024.

[101] Xiaohui Wu, Jia Wu, Xin Fu, Jian Li, Peng Zhou, and Xiaohua Jiang. Automatic knowledge graph construction: A report on the 2019 ICDM/ICBK contest. In *Proceedings of the 2019 IEEE International Conference on Data Mining (ICDM)*, pages 1540–1545. IEEE, 2019.

[102] Yilin Xiao, Junnan Dong, Chuang Zhou, Su Dong, Qianwen Zhang, Di Yin, Xing Sun, and Xiao Huang. Graphrag-bench: Challenging domain-specific reasoning for evaluating graph retrieval-augmented generation. *arXiv preprint arXiv:2506.02404*, 2025.

[103] Vikas Yadav and Steven Bethard. A survey on recent advances in named entity recognition from deep learning models. *arXiv preprint arXiv:1910.11470*, October 2019.

[104] Penghao Zhao, Hailin Zhang, Qinhan Yu, Zhengren Wang, Yunteng Geng, Fangcheng Fu, Ling Yang, Wentao Zhang, Jie Jiang, and Bin Cui. Retrieval-augmented generation for ai-generated content: A survey. *arXiv preprint arXiv:2402.19473*, 2024. Version 6.

[105] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 2025.

[106] Lingfeng Zhong, Jia Wu, Qian Li, Hao Peng, and Xindong Wu. A comprehensive survey on automatic knowledge graph construction. *arXiv preprint arXiv:2302.05019*, February 2023.

[107] Xiangrong Zhu, Yuexiang Xie, Yi Liu, Yaliang Li, and Wei Hu. Knowledge graph-guided retrieval augmented generation. *arXiv preprint arXiv:2502.06864*, 2025.

[108] Yuqi Zhu, Xiaohan Wang, Jing Chen, Shuofei Qiao, Yixin Ou, Yunzhi Yao, Shumin Deng, Huajun Chen, and Ningyu Zhang. LLMs for knowledge graph construction and reasoning: Recent capabilities and future opportunities. *arXiv preprint arXiv:2305.13168*, December 2024.

Appendix A: Code and Data Repository

The full source code developed for this thesis is available at: GitHub Repository: `https://github.com/nasibatuychieva/master_thesis-rag`

The repository contains:

- data preprocessing pipelines,
- chunked and embedded text representations,
- KG exports (nodes, edges),
- the hybrid RAG+KG retrieval implementation,
- the LLM-as-a-Judge assessment scripts.
- Gold Answer Dataset.