Universität Leipzig
Faculty of Mathematics and Computer Science

RAGnRoll - Towards Reconfigurable RAG Experimentation Without Overfitting

Master's Thesis

Jan Albrecht
Born Feb. 17, 1995 in Freital

Matriculation Number 3772326

1. Referee: Prof. Dr. Norbert Siegmund
2. Referee: Prof. Dr. Andreas Both

Submission date: May 15, 2025

Declaration

Unless otherwise indicated in the text or references, this thesis is entirely the product of my own scholarly work.

Leipzig, May 15, 2025

.....
Jan Albrecht

Abstract

Retrieval-Augmented Generation (RAG) systems enhance Large Language Models (LLMs) by grounding responses in external knowledge, addressing limitations like knowledge cutoffs and hallucination. However, developing and evaluating RAG systems is complex, involving numerous configurable components and risking overfitting during iterative refinement, often lacking systematic approaches. This thesis introduces RAGnRoll, a comprehensive benchmarking framework designed to facilitate systematic, reproducible, and generalizable RAG experimentation, ultimately aiming to prevent overfitting during configuration. RAGnRoll provides a structured methodology encompassing systematic baselining against standalone LLMs and naive RAG, a strict validation-test split for robust generalization assessment, multi-faceted evaluation including end-to-end and component-level metrics with hardware considerations, and tools for transparent experiment logging and tracing. The framework's utility was demonstrated through an experiment in software configuration validation, showcasing its capability to guide iterative development and performance optimization. RAGnRoll empowers researchers and practitioners to make data-driven decisions, moving beyond ad-hoc tuning towards a principled approach for building effective and reliable RAG solutions.

Contents

Chapter 1

Introduction

The year 2017 can be stated as the beginning of the interesting journey of artificial intelligence language models. With the publication of "Attention is all you need" from Vaswani et al. [77], the rapid development of language models and later large language models (LLMs) took off.

Today there is a variety of real world products that are using this technology, such as content generators like ChatGPT [57] or Claude [3], translators like DeepL [12] and Coding Assistants like Github Copilot [20]. The list can be expanded with technologies like sentiment analysis, question answering systems, market research or education systems. Open-source models are available for each of these technologies, providing strong alternatives to proprietary services.

The remarkable capacity of large language models has led to wide acceptance in society. However, LLMs have fundamental problems that cannot be solved with more training or larger models. Training models frequently is expensive, so daily training isn't feasible. Therefore, every new piece of information, such as election outcomes, weather updates, or sports results, that has occurred between the last training and the user prompt are unknown to the model. On top of that, models can only be trained on available data. Private information of users that might be relevant for the prompt are not considered in the generation process. LLMs also struggle with long-tail information, which occurs rarely in the training data [40].

When information is missing or underrepresented, outputs may deviate from user inputs, repeat previous outputs, or be made up by the LLM [93]. This technology is already used in many sectors with billions of customers, such as in marketing, retail and eCommerce, education, healthcare, finance, law, and media. It is crucial to develop systems that are as correct as possible.

The solution to potential missing information in training is to provide all necessary information to the LLM beforehand within the prompt, so that

the generator just has to construct a coherent text for the user. This can be achieved with so-called Retrieval-Augmented Generation Systems (RAGs), where the raw user prompt is used to retrieve relevant data from a database that is summarized and inserted into the prompt for the generator. This method overcomes many of the challenges LLMs face. Data can be accessed from private and up-to-date sources. The frequency of information occurrences no longer matters as long as the database includes it for retrieval. Having recent information no longer requires training the underlying model.

However, the adoption of RAG systems introduces significant overhead and complexity. The system requires additional steps between the prompt request and output. Most of those steps cannot be parallelized. This results in longer inference times and also leads to a more resource-intensive system. Next to the increasing infrastructure costs, developing and maintaining a RAG is more time consuming than developing a LLM, because the LLM is a part of the larger system. RAGs are not by default significantly better systems than pure LLMs as Simon [71] showed. These complex systems are highly sensitive to small configuration changes.

Therefore, important questions arise:

> "Is an advanced RAG system necessary for your use case, or is a standalone LLM sufficient?"

> "Is your RAG system the optimal one for your specific use case?"

The answers to these questions are hard to find, because one has to implement a RAG to test it on a specific problem and dataset. The scientific landscape of Retrieval-Augmented Generation Systems is a vast community with rapid development. Staying up-to-date with that research topic is time-consuming for companies and research departments.

The implementation of the RAG is just one part of the decision process. It is difficult to evaluate LLMs and all systems that are based on this technology, because outputs are not deterministic and such models are like a black box. It is not obvious why the model responds with a certain output. Another problem is that there is a whole set of potential correct answers, because text-based outputs can have many forms. Let's consider following example:

Question: *Is the evaluation process of LLMs an easy task?*
Answer 1: *The evaluation process of LLMs is not an easy task.*
Answer 2: *No, the evaluation process is a difficult task.*

Both answers are correct, but which metric must be used to measure this outcome?

The evaluation of RAGs is even harder because it has the same problems in addition to its own. RAGs are complex and are composed of many parts. Each part can lead to errors, and therefore all components of the system need to be evaluated in addition to an end-to-end evaluation of the whole system, as Salemi [68] and Yu [89] showed.

There are companies and research groups that successfully solved parts of this problem with developing tools, frameworks, and libraries such as AutoRAG [43], Llama-Index [51], LangChain [8], RaLLe [26], FlashRAG [39], RAGLAB [92], Haystack [60] and FastRAG [36]. It can be stated that all of these tools and frameworks are focused on developing RAG variants, making them production-ready, or evaluating them for performance, ignoring the fact that RAGs must be measured for hardware metrics such as latency, inference time, and CPU usage to determine if the benefits in performance compensate for the disadvantages. Additionally, Simon [71] showed there is a lack of external validity in the development of RAGs, because the iterative reconfiguration of these systems that leads to the best performance is a hyperparameter tuning process that might overfit the model to the seen data and therefore requires a dataset split with a validation dataset and a holdout test dataset, the latter of which is only used to estimate the generalization error.

In this master's thesis, we will make two contributions to the scientific landscape of RAGs: (i) A novel benchmarking framework, RAGnRoll, following the systematic blueprint shown by Simon [71] and evaluating hardware metrics alongside state-of-the-art (SOTA) performance evaluations, (ii) A practical demonstration of RAGnRoll through an experiment on the software engineering task of configuration validation. For (i), the framework will extend Haystack [60] and leverage the FastRAG library [36]. Haystack is an open-source framework for LLMs, RAGs, and SOTA search systems. FastRAG builds upon Haystack and adds special RAG architectures.

The primary results of this work indicate that while RAG systems offer a powerful approach to overcoming LLM limitations, their effective implementation is non-trivial. The development of RAGnRoll highlighted the critical necessity of incorporating rigorous machine learning practices, such as a strict validation-test split, into RAG evaluation to mitigate overfitting. Furthermore, the application of RAGnRoll to software configuration validation revealed that retrieval quality is often the predominant bottleneck, and complex RAG architectures do not automatically surpass well-prompted standalone LLMs without careful, task-specific optimization of the retrieval components. Our experiments achieved a competitive F1-score of 0.776 for configuration validation using a standalone model with a Ciri-like few-shot setup, underscoring the

importance of robust baselines.

This thesis concludes that a systematic and holistic benchmarking approach, as facilitated by RAGnRoll, is indispensable for advancing RAG technology. It enables data-driven decisions, helping to determine if a RAG system is truly necessary and how to best configure it for a specific use case, moving beyond ad-hoc tuning towards more principled and reliable RAG solutions.

This thesis is structured as follows: Chapter 2 provides an overview of the foundational concepts, including Large Language Models, text retrieval techniques, and the architecture of Retrieval-Augmented Generation systems. Chapter 3 discusses existing research and tools related to RAG evaluation and benchmarking. Chapter 4 details the design and methodology of the RAGnRoll framework, emphasizing its core principles for achieving generalizable and reproducible RAG experiments. Chapter 5 presents a practical application of RAGnRoll, detailing an experiment focused on using RAG for software configuration validation, and analyzes the obtained results. Finally, Chapter 6 summarizes the contributions of this thesis and discusses the implications of RAGnRoll for future research and development in the field of retrieval-augmented generation.

Chapter 2

Background

2.1 Large Language Models

LLMs are deep neural network models trained on large corpora of text data. They are predominantly based on the Transformer architecture [81]. Understanding Large Language Models and their hyperparameters such as temperature or top-p is crucial for the development of RAG applications. Below, we explain the Transformer architecture and its main components.

2.1.1 Transformer Architecture

There are many well-known models based on the Transformer architecture, such as BERT, GPT-3.5, LLaMA, and others [88]. The Transformer architecture was introduced by Vaswani et al. in 2017. It is based on the self-attention mechanism, which allows the model to weigh the importance of each word in a sentence. The architecture has been shown to outperform other architectures in various NLP tasks.

Figure 2.1 illustrates the Transformer architecture. Both encoder and decoder can be standalone models. Text must be tokenized through a dictionary to have a fixed size and frozen vocabulary. The encoder embeds a sequence of tokens and passes them to the positional encoding layer to preserve information about token position. Subsequently, it passes through multiple layers of multi-head self-attention and feed-forward neural network blocks. Attention heads compute token relevance via scaled dot-product attention: $softmax(QK^T/\sqrt{d_k})V$, where Q, K, and V are learned projections. The output of the encoder is a real-valued numerical vector and can be passed to the decoder. The decoder embeds an empty or predefined sequence. It gets passed to several layers of masked and non-masked multi-head self-attention and feed-forward neural network blocks. The mask ensures the decoder can

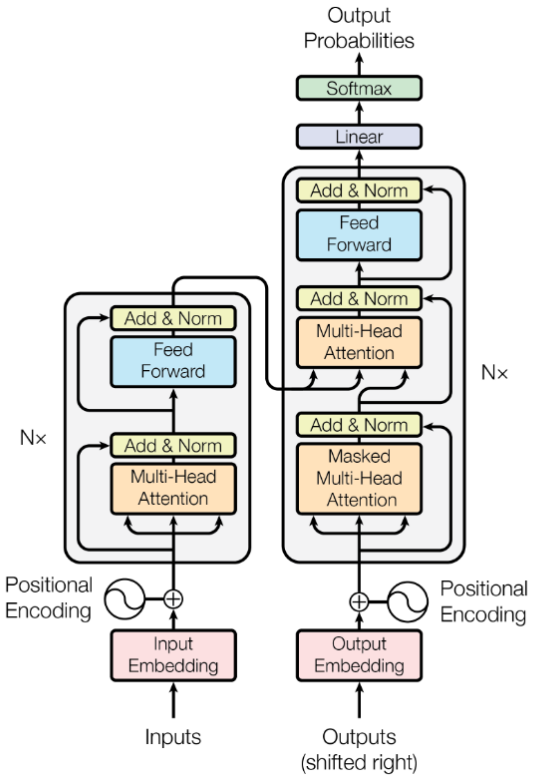

Figure 2.1: Transformer Architecture with encoder (left) and decoder (right), Source: Vaswani [77]

only use preceding tokens. The output probabilities forecast the next token in sequence. Therefore, prior decoder outputs can be used to autoregressively generate sequences of tokens.

2.1.2 Tuning Parameters

There are several tuning parameters that can be used to improve the performance of LLMs. Some of the most important tuning parameters are:

Temperature The temperature parameter is used to control the randomness of the generated text. A high temperature value leads to more randomness, while a low temperature value leads to less randomness. The temperature parameter T modifies the softmax function as follows:

$$softmax : o(z)_i = \frac{e^{z_i/T}}{\sum_{j=1}^{K} e^{z_j/T}}$$

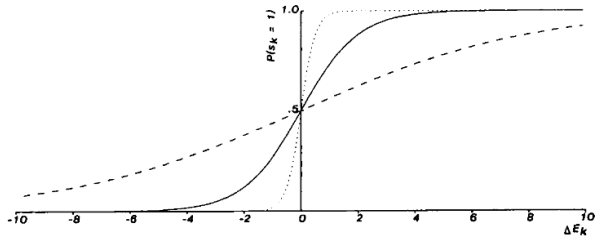

Figure 2.2: The softmax function with different temperature values, T=1.0 (solid), T=4.0 (dashed), T=0.25 (dotted), Source: Ackley et al.[1]

Figure 2.2 shows the softmax function. A low temperature value leads to a peaky distribution, while a high temperature value leads to a more uniform distribution and, therefore, to more randomness.

Top-K Sampling Top-K sampling [19] randomly selects from the top K tokens with the highest probability before applying the softmax function. Therefore, the higher the K value, the more tokens are considered for sampling. Higher values increase output randomness. With Top-K equal to 1, the model behaves like greedy decoding.

Top-P Sampling The Top-P sampling [24] randomly selects from the smallest set of tokens whose cumulative probability exceeds the probability P. Therefore, the higher the P value, the more tokens are considered for sampling. The outcomes get more random. It is a generalization of Top-K sampling, where the number of tokens to sample from is not fixed.

2.1.3 Prompting Techniques

The Transformer architecture is not deterministic as shown in previous sections. Additionally, LLMs also exhibit high variance in their outputs. The input prompt critically influences output quality. Therefore, there are numerous prompting techniques to stabilize desired outputs. Schulhoff created a taxonomy and recognized 58 LLM prompting techniques [69]. We discuss three representative prompting techniques: Few-Shot, Zero-Shot, and Chain-of-Thought (CoT). The list is not exhaustive, but these techniques are very commonly used and easy to implement.

Few-Shot vs. Zero-Shot Few-shot prompting is a form of In-Context Learning (ICL), in which the prompt includes examples of similar tasks to show the model how a task should be done. Brown showed that large models

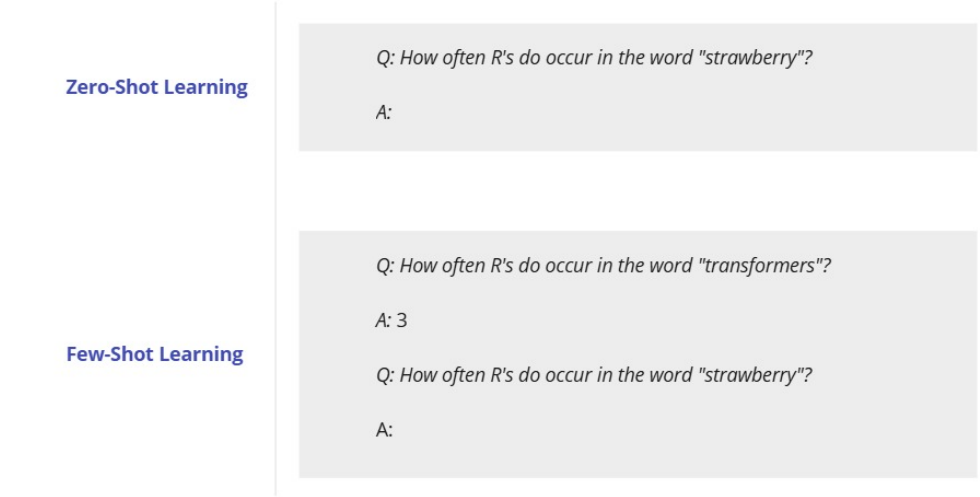

Figure 2.3: An example with a difficult query that many LLMs struggle with, first without an example (Zero-Shot) and then with one example (Few-Shot).

can leverage examples for various tasks [6]. Zero-shot prompting then refers to prompting without showing examples before the task. Even though few-shot prompting might increase the performance of LLMs, it also comes with increased costs, as there are more tokens to process. An example of these prompting methods can be seen in Figure 2.3.

Chain-of-Thought This method introduced by Wei leverages few-shot learning through simulating a written thought process in prior examples [79]. The LLM is therefore forced to adopt this behavior, which leads to more accurate outputs in reasoning and math problems. Schulhoff refers to it as the class of Thought Generation techniques [69]. Figure 2.4 shows an example prompt.

2.2 Text Retrieval

There are several types of text retrieval for RAG systems. Zhao classifies four retrieval techniques based on how retrieved information is passed to the generation process [94]. This thesis focuses exclusively on query-based retrieval techniques due to their prevalence and broad adoption. The other three types are latent-representative-based retrieval, logit-based and speculative retrieval.

Another categorization of text retrieval can be done by the type of retrieval. There are three main types of retrieval: sparse retrieval, dense retrieval and

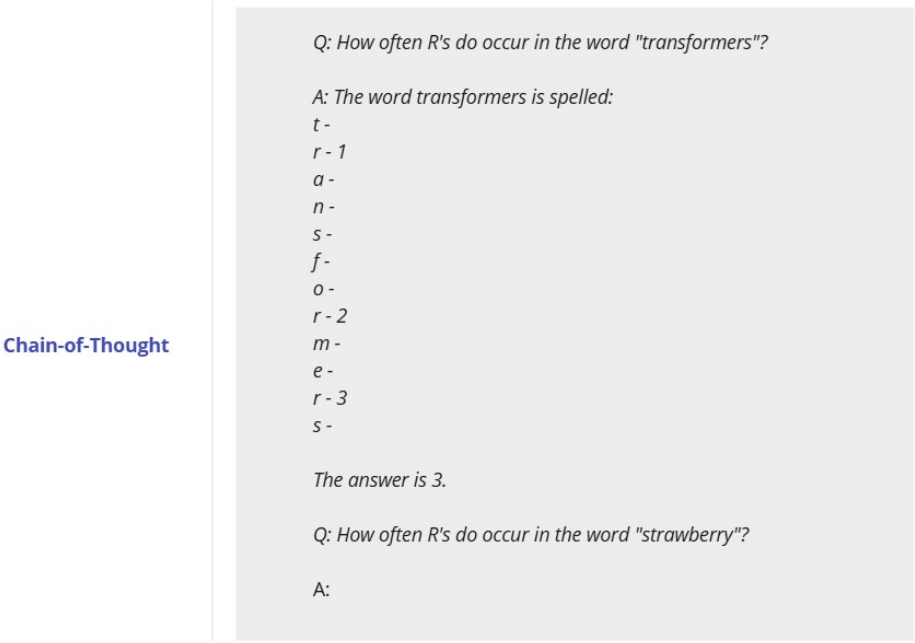

Figure 2.4: A Chain-of-Thought example with a difficult query that many LLMs struggle with without prompting techniques

hybrid retrieval. Sparse retrieval is based on traditional information retrieval techniques like TF-IDF or BM25. Dense retrieval is based on transformers models and hybrid retrieval combines both sparse and dense retrieval techniques. Following, we will describe these techniques in more detail.

2.2.1 Sparse Retrieval

TF-IDF and BM25 are widely implemented in libraries like LangChain [8] and LlamaIndex [51], representing standard sparse retrieval approaches.

TF-IDF TF refers to the term frequency of a term t in a document d. The inverse document frequency (IDF) is calculated as the logarithm of the total number of documents N divided by the number of documents containing the term t. Therefore the IDF factor is low for terms that occur in all documents and high for distinguishing terms that occur in frequently in a low number of documents. The position of the words is ignored [55].

$$\textit{TF-IDF}(t, d, D) = \textit{TF}(t, d) \cdot \textit{IDF}(t, D) = \textit{TF}(t, d) \cdot \log\left(\frac{N}{\textit{DF}(t, D)}\right)$$

BM25 Manning [55] presents multiple variants of BM25. A simplified version is defined as follows:

$$BM25(t, d, D) = IDF(t, D) \cdot \frac{(k_1 + 1) \cdot TF_{t,d}}{k_1((1 - b) + b \cdot \frac{L_d}{L_{ave}}) + TF_{t,d}}$$

The BM25 score is an advanced version of the TF-IDF score with two free parameters k_1 and b. The parameter k_1 is a scaling factor to determine how relevant term frequency is. The parameter b is for document length scaling, reducing scores of long documents.

2.2.2 Dense Retrieval

Dense Passage Retrieval as shown in Karpukhin [41] utilizes Bidirectional Encoder Representation from Transformers (BERT, [15]). BERT corresponds to the encoder component of a sequence-to-sequence Transformer architecture.Therefore it can be used to encode the text passages used as contexts at the classification token [CLS]. Text passages are mapped into a d-dimensional vector space under the assumption that semantically similar passages exhibit proximity. The query is also encoded into this space and the similarity between the mapped query vector and each passage vector is calculated. The similarity is calculated by the dot product or other similarity functions. The top-k similar passages are then selected and used for the generation.

Contemporary specialized embedding models are benchmarked through frameworks such as HuggingFace's MTEB [56]. While MTEB rankings suggest optimal embedding models, no universal solution exists. Embedding models are language- and domain-sensitive, as Gao [21] points out.

2.2.3 Hybrid Retrieval

Both sparse retrieval techniques TF-IDF and BM25 are good for keyword-specific searches, but perform poorly on a semantic comparison of documents and query. Encoders such as BERT, trained for language understanding, excel at semantic comparisons. Often, it is not trivial to determine whether a text retrieval task requires keyword-relevant sparse retrieval or dense retrieval that considers semantics.

Hybrid models compute both dense and sparse retrieval scores, combining them through weighted summation.

$$\alpha \cdot dense + (1 - \alpha) \cdot sparse$$

A factor $\alpha = 1$ results in using only dense retrieval. There is no universal performant value for α. This makes α a tunable hyperparameter requiring optimization.

2.3 Retrieval-Augmented Generation Systems

Large language models (LLMs) exhibit fundamental limitations that cannot be fully resolved through training or prompt engineering. Gao [21] stated that LLMs have significant limitations in knowledge-intensive or domain-specific tasks. Insufficient domain-specific training data often results in erroneous outputs. For large language models, this phenomenon is called *hallucinations*, as defined by Huang [29]. In retrieval-augmented generation (RAG) systems, hallucinations refer to outputs unsupported by provided context. Rashkin [63] defined hallucinations as any information that is neither inferable from nor stated in an external document. In this thesis, we will use the definition specific to RAG systems.

Factually incorrect output from an LLM can stem from several reasons. The information required to answer a question might be private, stored in a database that was not accessible during training. Additionally, the high cost of training/fine-tuning limits update frequency. This creates temporal gaps where post-training information remains unavailable. All these problems can be addressed if the LLM is not used to generate the answer itself, but rather to construct a coherent text passage based on a given question and its retrieved answer. Then a database would store all relevant information and would be updated as frequently as required. During inference, the system retrieves relevant document chunks from the database to inform answer generation.

This concept is very successful, as Shuster [70] showed. This approach reduces factual errors while enhancing open-domain conversational performance. Yu [89] concluded that this improves the reliability and richness of the produced content. Chen [9] identifies external knowledge integration as crucial for improving LLM accuracy and reliability.

There is a wide variety of architectures and approaches for retrieval-augmented generation models. In this section, we will give an overview of the most common approaches and architectures and start with the most basic ones. This section will follow the categorization of RAG systems by Gao [21], covering naive RAGs, advanced RAGs, a modular architecture, and special cases.

2.3.1 Naive RAGs

Naive RAGs represent the foundational implementation of retrieval-augmented generation. These systems concatenate retrieved context with user queries to guide LLM responses. The LLM is then only used for generating a coherent answer. Therefore, it is required to ingest relevant data for the given use case beforehand. The system then retrieves relevant chunks or documents of this data at inference time.

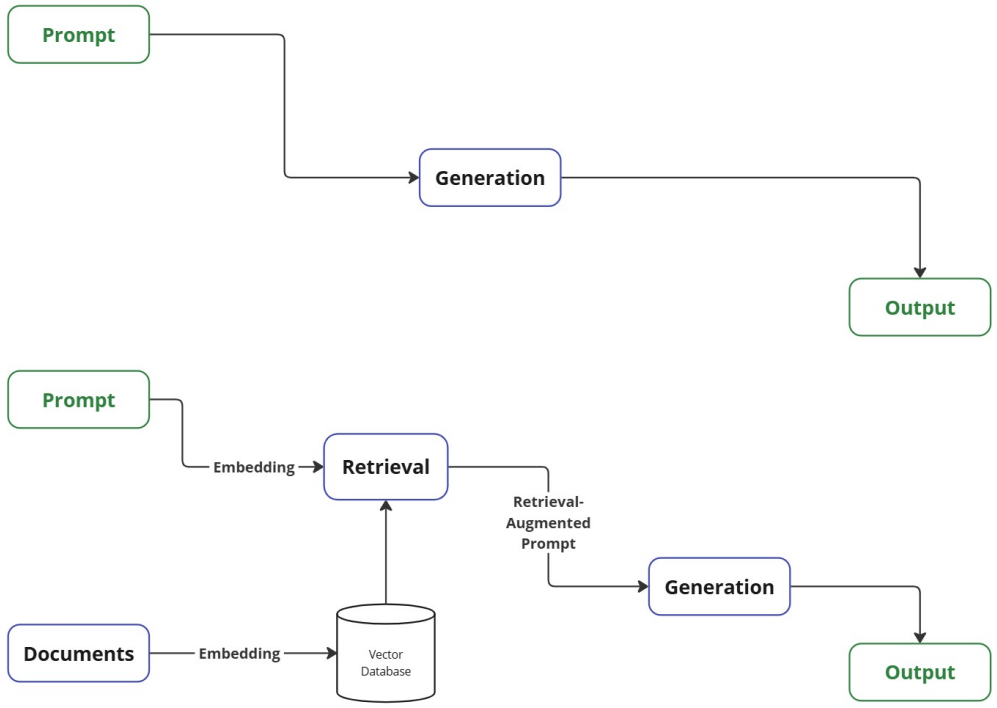

Figure 2.5: Comparison of the process: above, a standalone LLM takes a prompt and generates a response; below, a minimalistic RAG system performs embedding or indexing on a given set of documents and searches for the most similar documents for a given prompt at inference time. Both prompt and documents are used for the generation process.

This procedure can be seen in Figure 2.5. This generation phase is frequently termed the *read* operation in literature. Gao [21] defines the naive RAG system as *Retrieve-Read*. The retrieval can be done with sparse retrieval (TF-IDF, BM25), dense retrieval (DPR), or a hybrid version, as shown in Section 2.2.3.

Before the system can be used, it is necessary to ingest data. The inges-

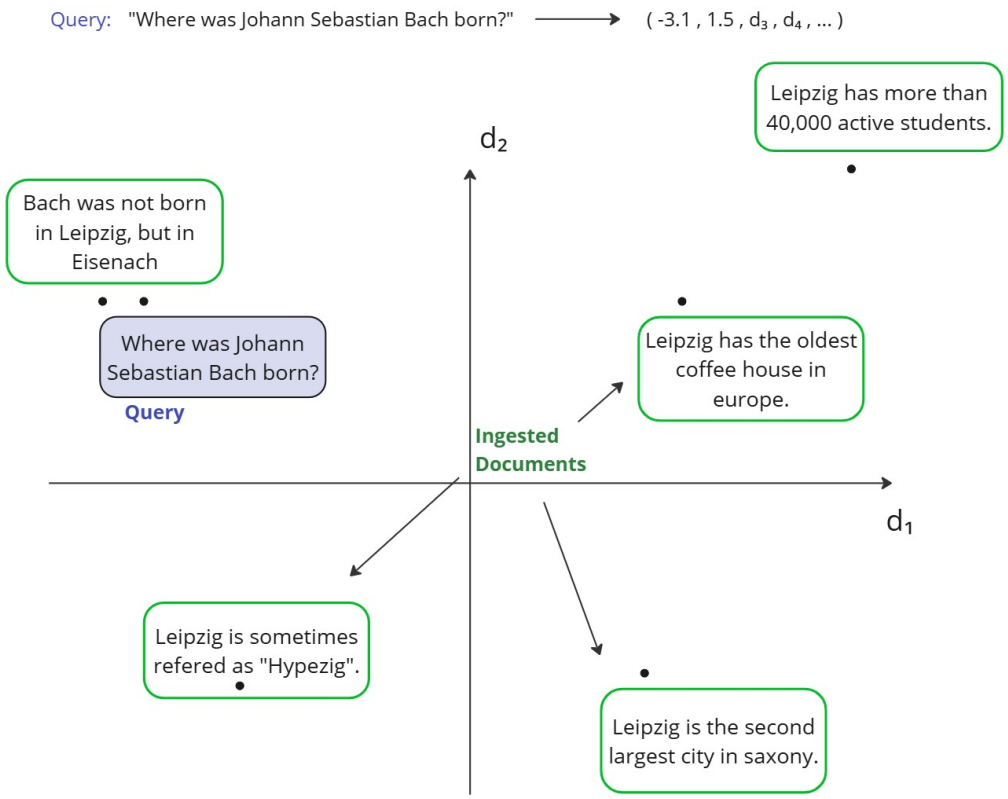

Figure 2.6: A vector space including mapped facts (documents) and a user query. The query and its corresponding correct answer are expected to have similar vectors

tion process includes preprocessing and selection of data that users are likely to need for their questions or use cases. Therefore, it is relevant to know the user base. It is neither feasible nor efficient to ingest all available data. The preprocessing pipeline converts raw data into document chunks using methods detailed in Section 2.3.2. As described in Section 2.2.2, the chunks are then embedded into a d-dimensional real-valued vector space. A simplistic minimal example of a vector space is shown in Figure 2.6. The documents get mapped to different locations within the space. This assumes semantic alignment between queries and relevant documents in the embedding space. In reality, there will be more documents close to each other, documents and queries will be longer or more complex, and each query will return the top-k chunks or documents, which are not always as relevant as in this example. Top-k can be seen as a hyperparameter for the system. The vectors are then stored in

vector databases, specialized databases for vector representations.

Gao [21] listed several drawbacks for naive RAGs. The basic retrieval suffers from insufficient recall and precision scores, leading to irrelevant documents, missing context, and bias. The integration of the provided context is a challenging process. The generator often overrelies on the augmented information, by simply repeating the retrieved content and failing to provide insightful conclusions. Therefore, this simplistic form of RAG requires advanced techniques to overcome these issues.

2.3.2 Advanced RAGs

There is no strict definition of advanced versions of retrieval-augmented generation systems. The term describes a loose bundle of techniques to improve the quality of such systems. A list follows, the items of which we will explain in greater detail subsequently. We will follow the survey definition by Gao et al. [21] of advanced RAG, which defines it as the *Rewrite-Retrieve-Rerank-Read* (4R) structure with chunking enhancements. We will skip *retrieve* and *read* here, as we described them in Section 2.3.1 in detail.

Chunking Chunking's importance becomes clear through a practical example: Let us use a sentence about Leipzig from Wikipedia [80]. In the "Music" section, the following sentence appears.

> "Johann Sebastian Bach spent the longest phase of his career in Leipzig, from 1723 until his death in 1750, conducting the Thomanerchor (St. Thomas Church Choir), at the St. Thomas Church, the St. Nicholas Church and the Paulinerkirche, the university church of Leipzig (destroyed in 1968)."

If there are thousands of Wikipedia pages or websites to process, then this content cannot be split manually. What constitutes a document or a good chunk in this context? If the query asks whether Johann Sebastian Bach lived in Leipzig, then embedding the whole sentence would not guarantee a high similarity vector score in the retrieval process. This differs from domain to domain. While chunking facts from sentences might be a valid strategy for Wikipedia, processing internal contracts in a large company would require less granular chunking.

Therefore, there are several chunking techniques to consider for tuning the RAG system. Fixed-size chunking divides text into uniform segments (e.g., 400-character blocks), while sliding window variants add overlapping regions. The sliding window approach adds an overlap of a few characters. Semantic chunking splits text based on predefined characters such as "$\backslash n$, $\backslash\backslash n$, *or*

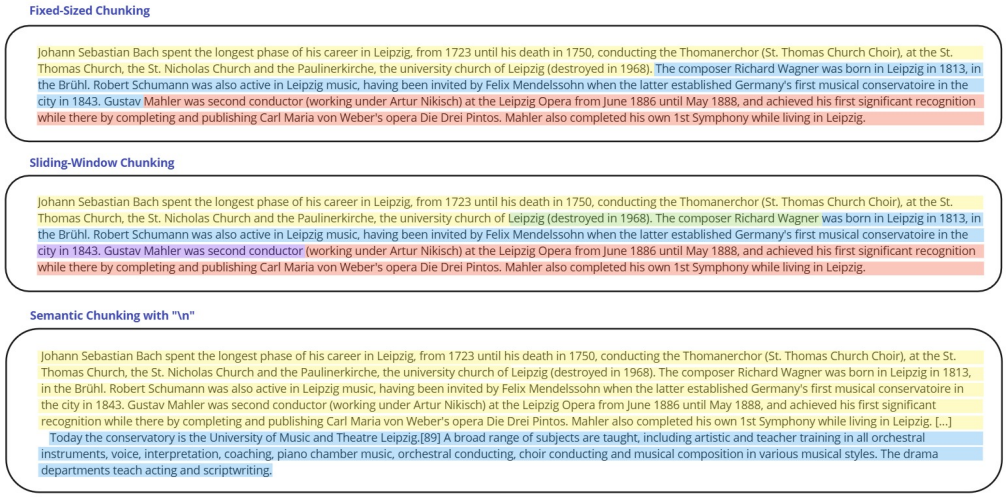

Figure 2.7: Different types of chunking shown on the example of a Wikipedia page of Leipzig [80]

*
*". These techniques are visualized in Figure 2.7. There are also special use-case techniques, such as Markdown, JSON, HTML, and programming code chunkers. Almost all presented chunking techniques are offered in typical libraries, e.g., Llama-Index [51] or Langchain [8].

Rewrite *Rewrite* is a collection of pre-retrieval techniques to increase the likelihood of relevant retrieved documents and a concise generation. The query can be transformed, expanded, or routed. Query routing, in this context, refers to directing queries to different RAG pipelines based on the query's nature [21].

Rerank *Reranking* mitigates risks from which naive RAGs suffer. The key consideration for retrieval is deciding how many documents should be retrieved. In real-world scenarios, there will not be perfect retrievals. Therefore, the retrieval process will always introduce noise in the form of irrelevant documents or chunks into the generation process. Additionally, if an insufficient number of documents are retrieved, relevant ones that do not appear at the top of the similarity score might be lost. Post-retrieval techniques, such as reranking, mitigate this risk by ranking and ordering these documents [21].

2.4 Modular RAG

There is a variety of RAG systems with many different components. The definition of advanced RAG systems is too strict and sequential to encompass this

variety. Modular RAGs close this gap. Modular RAGs can be seen as individual component blocks that can be added to a pipeline as long as the input and output of these blocks are compatible. Gao et al. introduced this definition and defined more components than the here-presented *rewrite, retrieve, rerank, and read* [21]. One simple example illustrating the need for modular RAGs is a routing component that decides whether retrieval is necessary at all. Redirecting to the generator can lead to reduced computational time and better results in some cases [54]. This modularity is a key requirement for the ability to build custom RAG systems for each use case.

2.4.1 Drawbacks of RAGs

Retrieval-augmented generation systems offer several improvements compared to standalone LLMs, as described at the beginning of this chapter. Nevertheless, this approach comes with drawbacks that must be considered before replacing standalone LLMs in production.

1. RAGs have significantly longer computation times and must be partially computed in sequential order. Therefore, the time-to-first-token (TTFT) is always higher than for a vanilla LLM.

2. This increase in computation times, and the fact that there are more LLM requests (e.g., for rewriting or reranking), result in higher costs.

3. RAGs introduce an overhead of work and require additional expertise for developing, maintaining, and experimenting with them.

4. RAGs can introduce a wide variety of diverse failures in all components of the system; failures can also occur during ingestion.

These points show that RAGs are not a free lunch, and it must be considered whether a RAG approach is needed for every specific use case. Therefore, there is a need for efficient RAG experimentation that can address all points necessary for making this decision.

Chapter 3

Related Works

3.1 RAG Evaluation

Difficulties in RAG/LLM Evaluation Yu, Gan, et al. [89] note the temporal complexity of information and the need for holistic evaluation methodologies in RAG systems. Ru, Qiu, et al. [65] and Chang, Wang, et al. [7] advocate for specialized evaluations over uniform benchmarks and highlight challenges such as dynamic out-of-distribution evaluation and data contamination. Zhao, Zhang, et al. [94] identify key limitations in RAG systems, including knowledge updates and data leakage, proposing research directions to address these issues. Gao, Xiong, et al. [21] stress the need for refined evaluation methodologies to keep pace with RAG's rapid evolution and expansion into multimodal domains.

Component-Specific Challenges and Failures in RAG Systems Li et al.[48] investigate various design choices in RAG systems, highlighting the superior performance of approaches such as Contrastive In-Context Learning RAG and Focus Mode RAG. They emphasize the critical importance of retrieved context quality and prompt formulation over simply increasing the knowledge base size. Liu et al.[52] note the difficulty in locating problems within the RAG pipeline, while Barnett et al.[5] identify seven key failure points in RAG systems, stressing the need for effective validation during operation. Huang et al.[29] provide a comprehensive taxonomy of hallucinations in LLMs, discussing their causes and mitigation strategies, and highlight the limitations of current retrieval-augmented systems. Zhao et al.[94] explore retrieval quality issues, suggesting that excessive retrieval can degrade results. Liu et al.[53] demonstrate that LLMs perform best when relevant information is positioned at the beginning or end of the context, underperforming when it is in the middle. Together, these studies underscore the multifaceted challenges

at the component level and illustrate that both design choices and operational failures contribute significantly to the overall limitations of RAG systems.

Evaluation Methodologies and Transparency Simon et al.[71] propose a blueprint for reusable empirical study designs for evaluating RAG systems. They emphasize the need for open data for reproducibility while ensuring it remains closed to LLM training to prevent data leakage. The authors also stress the importance of standardized methodologies and transparent reporting to ensure validity and replicability. Wagner et al.[78] underscore the challenges in reproducing LLM results due to unknown hyperparameters and training data, advocating for transparent yet encoded datasets to enhance reproducibility. Pimentel et al.[61] highlight the impact of implementation details on model performance and the need for transparent and standardized reporting of metric calculation procedures to foster reproducibility and comparability across studies. They emphasize that LLM performance is highly dependent on the evaluation methodologies and implementation details used. Overall, these works call for a shift towards greater transparency and standardization in evaluation practices, which is essential for robust performance assessments in RAG systems.

End-to-End Evaluation Traditionally, RAG evaluation has primarily relied on end-to-end assessment by comparing the generated output with one or more ground truth references [68]. Although this approach is crucial for gauging overall performance, it suffers from significant limitations when evaluating retrieval models. In particular, end-to-end evaluation lacks transparency regarding which retrieved document contributed to the final output, thereby hindering the interpretability of the system's behavior.

While end-to-end evaluation provides a holistic view of RAG system performance, understanding the contributions of individual components to the overall system behavior is equally critical. In their comprehensive study, Li et al. [48] systematically investigate the intricate relationships between various RAG components and configurations, addressing nine key research questions to unravel the operational mechanisms of RAG systems.

Component Evaluation The evaluation of generation tasks, especially those involving creativity or open-ended responses, is inherently challenging due to the subjective nature of defining "correct" or "high-quality" outputs [89]. To address this, researchers have proposed advanced evaluation methods. For instance, an "Oracle Prompt," which includes all relevant documents, serves as an upper bound for evaluating generation by simulating perfect retrieval conditions [44]. However, this method can overemphasize retrieval quality

by assuming ideal conditions. Li et al. emphasize the widespread adoption of LLMs as judges, noting their ability to align with human judgments while highlighting significant biases, such as preference leakage towards related models [48].

Evaluating the retrieval component of RAG systems presents unique challenges. Ashkan Alinejad et al.[4] highlight the complexity of assessing retrieval effectiveness by comparing human judgments with four evaluation types: Exact Match, Token-Based, Embedding-Based, and LLM-Based. Alireza Salemi and Hamed Zamani [68] introduce eRAG, a novel evaluation framework that leverages downstream task performance to assess retrieval quality more effectively. They demonstrate that traditional relevance labels have a weak correlation with RAG performance, emphasizing the need for task-oriented evaluation methods. Jin et al.[37] reveal that increasing the number of retrieved passages does not consistently improve end-to-end performance, often leading to performance degradation due to hard negatives. To mitigate this, they propose methods like retrieval reordering and fine-tuning techniques to enhance context utilization. Furthermore, Mallen et al. [54] find that retrieving contexts may be unnecessary and even detrimental when dealing with common knowledge, but it benefits questions about rare knowledge, underscoring the importance of contextual relevance in retrieval strategies.

3.2 Frameworks

Recent advancements in frameworks for RAG development and evaluation have significantly enhanced the systematic assessment of Retrieval-Augmented Generation (RAG) systems. RAGAS [16] introduces reference-free metrics to evaluate faithfulness, answer relevance, and context relevance, while InspectorRAGet [18] provides an interactive platform for comprehensive analysis of RAG systems' performance and annotator quality. RaLLe [25] focuses on developing and optimizing R-LLMs through transparent prompt engineering, whereas RAGGED [28] offers insights into model behaviors under varying context configurations. FlashRAG [38] and BERGEN [64] are modular toolkits supporting diverse RAG components, datasets, and metrics, with FlashRAG emphasizing efficiency and BERGEN promoting reproducibility. AutoRAG [42] automates pipeline optimization, RAGCHECKER [65] employs claim-level metrics for fine-grained evaluation, and ARES [67] leverages synthetic data and lightweight judges for scalable assessment. R-Eval [76] and RAGLAB [91] provide unified, customizable frameworks for evaluating domain knowledge and comparing RAG algorithms, respectively. Although these frameworks significantly advance RAG research, they do not explicitly address the challenge of

overfitting to the validation dataset, which can arise from the iterative reconfiguration of RAG systems. This problem is comparable to that encountered when training large language models if there is no training split into train, validation, and holdout test datasets. In the following chapter, we will introduce an evaluation design that enhances the generalization of such systems in a transparent and reproducible manner.

Chapter 4

Designing Generalizable and Reproducible RAG Experiments

Developing retrieval-augmented generation systems is a challenging task that often requires multiple reconfiguration phases [71]. Because RAG systems can involve complex pipelines with iterative or recursive processes, component evaluation and in-depth failure analysis are crucial for tuning the appropriate components. Failures can occur throughout the RAG system as highlighted by Barnett et al. [5]. Since every additional component can influence overall performance, it is indispensable for a robust evaluation framework to assess individual components alongside end-to-end system results. This chapter addresses these challenges by first introducing a validation-test split for evaluation data and justifying its importance. Subsequently, we explore accurate RAG evaluation from two perspectives: end-to-end and component-level assessment. We then delve into methods for accelerating RAG development while ensuring transparent and reproducible results, including Haystack's approach to modular development via configuration files, extended here to support multi-configuration testing. Finally, we cover failure analysis using tracing and the estimation of generalization error.

4.1 Validation-Test Split

Typical machine learning projects require researchers to collect, prepare, and split data into training, validation, and test sets. This split is crucial for optimizing a model using the training and validation datasets, and subsequently testing its generalization error on the unseen test dataset. The necessity for this process stems not solely from training itself but primarily from tuning models to a given dataset - often involving numerous free parameters (hyper-parameters). Whenever tuning parameters based on performance on a specific

dataset, it is essential to ensure that the model does not overfit this data. We argue that this challenge is equally present in the development of RAG systems.

Whether training a classifier or configuring a RAG system, both processes involve a large number of free parameters. A RAG system can have hundreds to thousands of them, especially when considering choices for generator models, embedding techniques, corpus content, or system architectures as (indirect) free parameters. Therefore, it is crucial to ensure that the final system configuration is not overfitted to the specific evaluation dataset used during development.

In this framework, we address this by splitting any given evaluation dataset via random sampling into two parts: a validation dataset and a test dataset. We then proceed with evaluation and reconfiguration using *only* the validation dataset, holding out the test data completely until a satisfactory configuration is achieved through iterative refinement. Once the reconfiguration phase is complete, we use the test set *once* to assess whether the performance observed on the validation set generalizes, typically by comparing the validation error (or other metrics) with the test error. The next section explains how RAG systems should be evaluated to maximize their overall performance using this approach. If the validation error is, for some metrics, significantly lower than the test error, then there is a chance of overfitting. We do not perform hypothesis tests within our framework.

4.2 Evaluation Techniques

Evaluating retrieval-augmented generation systems is a challenging task and an active area of research. It inherits common machine learning evaluation challenges, such as data shifts, generalization errors, and data contamination, but also introduces unique failure points due to the complexity of its multi-component design. In this section, we will discuss major failure points specific to RAG systems and explain how this framework aids in their identification. The ultimate goal when tuning a RAG system is to maximize its performance in end-to-end evaluation, ensuring that the system's responses fully and correctly answer user questions or complete assigned tasks.

However, relying solely on end-to-end evaluation makes it difficult to pinpoint which parameters to tune or which data modifications are needed to achieve performance improvements. Figure 4.1 illustrates example failures for components in a specific RAG pipeline, but in practice, the potential points of failure are far more numerous. RAG systems operate like complex processing pipelines; identifying and replacing underperforming bottleneck components

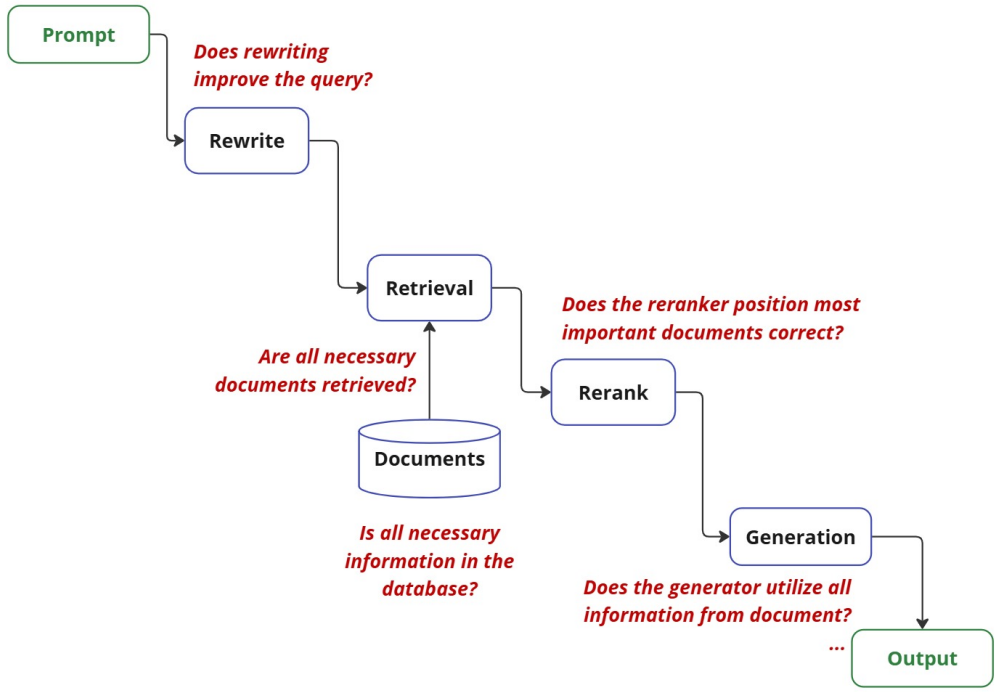

Figure 4.1: A RAG pipeline with one failure example for each used component.

with more effective ones is essential for optimization. This necessitates rigorous failure analysis. Therefore, every experiment should incorporate tracing mechanisms to track the flow of information and understand the root causes when a query is answered incorrectly. In this framework, we utilize Langfuse's [34] self-hosted version for tracing. We visualize all metrics and RAG parameters using MLflow [46]. While MLflow also offers tracing and GenAI evaluation modules, we currently opted against using them because its tracing module lacks compatibility with Haystack at present, and its GenAI features require ground-truth documents (which can be impractical, e.g., due to dynamic chunking strategies), are marked as experimental, and are incompatible with our validation-test split methodology.

4.2.1 End-to-End Evaluation

We define end-to-end evaluation as metrics that use the system's input query and its final output, comparing this output against a ground-truth reference to determine correctness. Evaluations that assess intermediate steps, such as retrieval performance or the generator's ability to utilize provided context, are

Metric	**Formula / Description**
Accuracy	$\frac{TP+TN}{TP+TN+FP+FN}$
Precision	$\frac{TP}{TP+FP}$
Recall	$\frac{TP}{TP+FN}$
F1-Score	$2 \times \frac{Precision \times Recall}{Precision+Recall}$
Matthews Correlation Coefficient	$\frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$
False-Positive Rate	$\frac{FP}{FP+TN}$
False-Negative Rate	$\frac{FN}{FN+TP}$

Table 4.1: Typical classification metrics used for experiments involving RAGs or LLMs[27, 90].

considered component-level evaluations and are distinct from the end-to-end perspective discussed here. In this thesis, our focus is primarily on classification tasks; therefore, the end-to-end metrics employed are limited to those suitable for classification.

In stark contrast to evaluating open-ended generative tasks, classification tasks benefit from well-established metrics and evaluation methods. Table 4.1 presents commonly used classification metrics relevant for RAG and LLM experimentation.

Meaningful end-to-end evaluation requires researchers to establish *baselines* for their experiments. Baselines make results interpretable by providing essential points of comparison. Therefore, this framework implements two default baselines used when initiating an experiment. The first baseline consists solely of a standalone LLM answering the query without retrieval. The second is a naive RAG system employing BM25 retrieval with data from the predefined corpus (a simple Retrieve-Read pipeline, cf. Section 2.3.1). The standalone LLM baseline helps justify the complexity overhead of implementing a RAG system; if an evaluated RAG system cannot surpass the performance of the LLM baseline, a simpler approach might be preferable. Advanced RAG systems often involve more components, potentially leading to longer computation times and increased costs. Outperforming the naive RAG baseline demonstrates the value added by the more advanced RAG configuration be-

yond simple keyword retrieval.

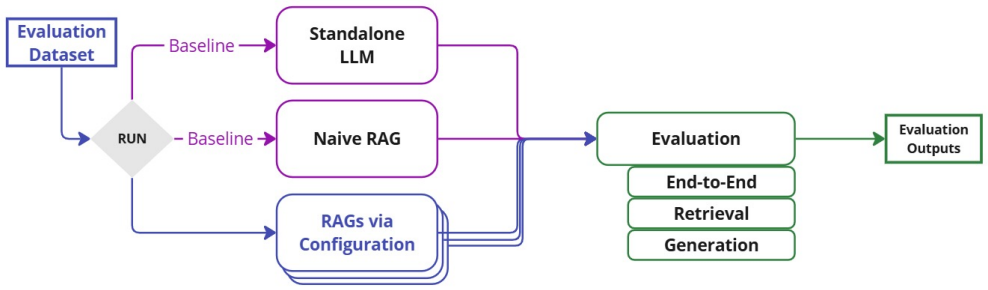

Figure 4.2: Framework Overview: Researchers define configuration files for RAG variations, which are evaluated against the standalone LLM and naive RAG baselines using end-to-end metrics.

Figure 4.2 illustrates the state of our framework as defined thus far and clarifies our approach. It depicts the process where evaluation data is used to compute metrics for both the defined baselines and the RAG configurations under test, performing both end-to-end and component-wise (retrieval and generation) evaluations. The following section details the component evaluation methodology employed in our framework.

4.2.2 Component Evaluation

Component evaluation is necessary to detect performance bottlenecks or issues introduced by individual components or problematic interactions between them [68]. In Figure 4.1, we presented a few examples of such failures. While illustrative, this is far from an exhaustive list of potential failure points in a complex RAG system. We argue that potentially *all* components, no matter how seemingly small their impact, can contribute significantly to the overall system's failure rate. Therefore, ideally, it would be necessary to evaluate them all component-wise.

However, current literature on component evaluation primarily focuses on the core retrieval and generation (*read*) stages. Furthermore, we want to emphasize that evaluating components in complete isolation from others is often impractical. If, for example, one wished to evaluate the generator component independently of the retrieval step, it would require providing a ground-truth dataset of ideal retrieved contexts for each query. This might be feasible for simple scenarios requiring only a single, easily identifiable document chunk (e.g., a sentence providing a direct answer). Yet, real-world scenarios frequently involve more complex cases, such as ambiguous queries, multifaceted

questions requiring information synthesis, or queries that do not require retrieval at all [29]. Furthermore, if the researcher modifies the chunking strategy (e.g., size or technique), the ground-truth contexts would need to be recreated, as the previously selected chunks might no longer exist (cf. Section 2.3.2).

Considering other components besides the retriever, such as query rewriters or document rerankers, establishing ground-truth data becomes even more problematic. The purpose of rewriting a query, for instance, is to improve subsequent retrieval performance. Thus, its effectiveness is inherently dependent on the specific retriever used and cannot easily be assessed against fixed, ground-truth examples in an evaluation dataset.

Given these challenges, the primary purpose of component evaluation shifts towards identifying weaknesses within the RAG pipeline rather than achieving perfect isolated assessment. It is often more feasible to evaluate the RAG system using only the initial inputs and final ground-truth answers for end-to-end metrics, while employing non-deterministic LLM-as-a-Judge methods for assessing intermediate steps where appropriate. The LLM-as-a-Judge approach involves using a powerful LLM (often significantly larger or specifically trained for evaluation) to assess the quality of intermediate outputs (such as retrieved context relevance) or final responses [10]. Instead of relying solely on numerical metrics, the judge LLM provides qualitative assessments or scores based on predefined criteria.

Relying primarily on system input and final ground-truth datasets for evaluation offers significant time savings compared to creating extensive intermediate ground truths. However, detailed failure analysis to pinpoint the root cause of errors still necessitates tracing individual pipeline executions, which we utilize in this framework with Langfuse [34]. The following sections detail our approach to evaluating the key retrieval and generation blocks using a combination of these techniques.

Retrieve Evaluating the retrieval component is arguably one of the most challenging aspects of RAG assessment for several reasons. First, retrieving more documents does not necessarily imply better final results, and it is often unclear which specific set of documents will best enable the generator to produce the optimal answer [38]. Second, real-world indexed corpora often contain redundant and contradictory information [89]. Third, some queries require retrieving diverse perspectives on a topic, highlighting various considerations. Fourth, the quality of the input query itself significantly affects retrieval performance; queries can be overly complex or ambiguous, or may not even require retrieval, potentially leading to generator hallucinations if irrelevant or misleading context is fetched [29, 54].

While retrieved documents can often be assessed in a binary fashion (rele-

vant/irrelevant), allowing for metrics based on *True/False Positives/Negatives*, simple metrics like precision and recall are often not sufficiently sensitive to capture nuanced differences in retrieval quality [89]. There is a clear need for evaluation techniques that also consider the ranking of retrieved documents.

Furthermore, significant limitations exist when working with ground-truth data for retrieval evaluation. Defining a definitive ground-truth set of relevant context documents for a given query can be ambiguous and challenging even for human experts, although LLM-as-a-Judge models have shown high correlation with human judgments in this area [10]. Compounding this issue is the dynamic nature of chunking during reconfiguration. Every time a chunking technique or parameter is altered, the resulting indexed documents change, necessitating the recreation of ground-truth retrieval datasets. This poses a significant bottleneck for rapid development cycles, yet tuning chunking strategies is crucial. Therefore, to maintain flexibility and speed, this framework focuses primarily on LLM-as-a-Judge evaluation for the retrieval component, particularly enabling effective chunking parameter tuning.

We utilize a metric named *Context Relevance* (akin to *Context Precision* in other frameworks) which employs an LLM-as-a-Judge. This metric assesses the relevance of each retrieved document concerning the input query. A binary approach is typically adopted: the LLM-as-a-Judge determines if a document contains statements relevant to the query. If yes, the document scores 1 (relevant); otherwise, it scores 0 (irrelevant). Haystack's 'ContextRelevanceEvaluator' [60], for instance, implements such a binary assessment. The final *Context Relevance* score is the mean of these binary scores across all retrieved documents for a query (and typically averaged over all queries in the dataset), yielding a score analogous to Mean Precision.

Recognizing that simple relevance counts (like Context Relevance/Precision) do not capture ranking quality, we also implement *Mean Average Precision at K (MAP@K)* to offer a complementary, rank-aware perspective. MAP@K evaluates whether relevant documents are ranked higher in the retrieved list [17].

$$MAP@K = \frac{1}{|Q|} \sum_{q=1}^{|Q|} AP@K$$

$$AP@K = \frac{1}{N} \sum_{k=1}^{K} Precision(k) \cdot rel(k)$$

- $|Q|$ is the total number of queries in the evaluation set.
- q represents a specific query

- $AP@K$ is the Average Precision at K for a specific query.
- N is the total number of relevant items for a particular query.
- K is the number of top documents being evaluated.
- k represents a specific position in the ranked list of retrieved documents.
- $Precision(k)$ is the precision calculated at position k, defined as the number of relevant items among the top k items divided by k.
- $rel(k)$ equals 1 if the item at position k is relevant and 0 otherwise.

Mixed Results

Doc K	Rel(K)	AP@K Term
Doc 1	1	1/3
Doc 2	0	0
Doc 3	1	2/9
Doc 4	0	0
Doc 5	0	0
Doc 6	1	1/6

AP@K = 1/3 + 2/9 + 1/6 =13/18

Best Case

Doc K	Rel(K)	AP@K Term
Doc 1	1	1/3
Doc 2	1	1/3
Doc 3	1	1/3
Doc 4	0	0
Doc 5	0	0
Doc 6	0	1/6

AP@K = 1/3 + 1/3 + 1/3 = 1

Worst Case

Doc K	Rel(K)	AP@K Term
Doc 1	0	0
Doc 2	0	0
Doc 3	0	0
Doc 4	1	1/12
Doc 5	1	2/15
Doc 6	1	1/6

AP@K = 1/12 + 2/15 + 1/6 = 23/60

Figure 4.3: An example of the AP@K metric for a single query. For each retrieved document, the relevance and precision from the first to the K-th document are averaged across all retrieved documents.

The formula presented is the standard definition of MAP@K [50]. While in some traditional information retrieval scenarios K might be set very high, this can be computationally expensive and may conflict with the goal of providing concise context. Therefore, we calculate MAP@K based on the actual number of documents retrieved (i.e., the system's configured *Top-K* parameter). *Context Relevance* measures overall retrieval quality by focusing on the proportion of relevant items, while *MAP@K* specifically assesses the ranking quality within the retrieved set. A low *Context Relevance* score alongside a high *MAP@K* score might indicate that while relevant documents are found and ranked well, too many irrelevant documents are also being retrieved, suggesting the *Top-K* parameter could potentially be reduced. Conversely, a low *MAP@K* suggests the retrieval/ranking mechanism itself needs adjustment, as irrelevant documents frequently appear high in the results list.

Generators Generators in RAG systems, typically based on transformer architectures, were not initially designed primarily for classification tasks. Consequently, ensuring their outputs adhere to specific, desired formats (e.g., producing only *valid* or *invalid*) is an important preliminary check. This necessitates a metric or process to evaluate the generator's ability to follow formatting instructions; we refer to this as *format validation*. However, a significant challenge arises when evaluating generators solely on classification accuracy: simple binary outputs (*True/False*, *Valid/Invalid*) make it difficult, if not impossible, to directly assess crucial underlying aspects like whether the retrieved context was properly utilized or how relevant the generated answer is to the context.

Even when the final prediction is simply *True* or *False*, the underlying generation process involves varying degrees of context utilization and reasoning quality. Therefore, in this framework, we implement a structured output approach for classification tasks to gain more insight. Rather than having the generator produce *only* the binary label (e.g., *0/1*, *valid/invalid*, *True/False*), we utilize an AnswerBuilder component that prompts the model to provide its reasoning *alongside* the classification verdict. This component ensures that the final answer label (e.g., the phrase *'The answer is "True"'*) appears in a consistent, predictable format within the output, facilitating reliable extraction for end-to-end metric calculation. The AnswerBuilder can be configured to support various output patterns while maintaining compatibility with automated evaluation processes. Figure 4.4 illustrates this structured output approach.

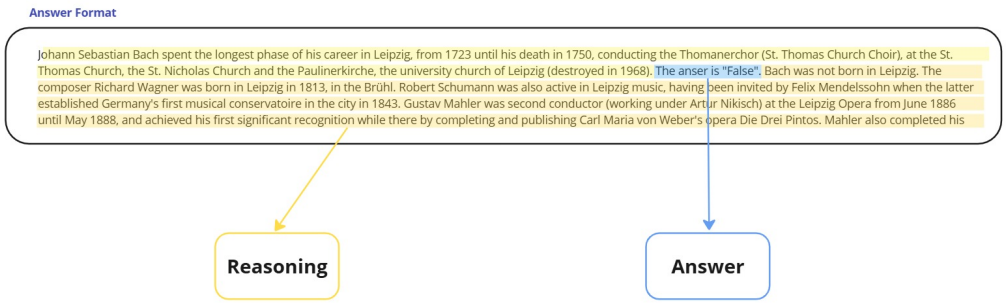

Figure 4.4: Example of a generator's answer, including reasoning, and the final extracted answer used for binary classification.

This approach has several positive side effects. First, the reasoning can be measured to assess context utilization or answer relevance, which is important for debugging the system. If the retrieved context is not sufficiently utilized, this issue might not be identified by analyzing only binary classes. Second, it enables researchers to leverage the current test-time compute paradigm shift

for large language models or *large reasoning models*. In practice, these models use *<think>...</think>* tags and various search algorithms to find the optimal reasoning path before answering the query, leading to significantly better results [84].

4.2.3 Component Block Evaluation

Beyond evaluating individual core components like the retriever and generator, we must also consider components whose performance is highly interdependent or difficult to assess in isolation. Examples in advanced RAG systems include query rewriters and document rerankers. Rewriters aim to optimize the input query for the specific retrieval method and available context. The optimal rewriting strategy often depends on whether the subsequent retriever is sparse (keyword-based) or dense (embedding-based), requiring optimization towards relevant keywords or semantic similarity, respectively. Rerankers, operating post-retrieval, aim to reorder the retrieved documents to prioritize the most relevant or useful ones for the generator.

Both rerankers and generators significantly influence, and rely upon, effective context utilization – the ability to synthesize a coherent and correct answer based on facts within the retrieved context. However, interactions can be complex. For instance, phenomena like 'lost-in-the-middle', where information in the middle of long contexts is less effectively utilized, significantly impact performance, and different LLMs exhibit varying susceptibility to this effect [53]. Consequently, combining the individually 'best' performing reranker and generator (based on isolated metrics) might not yield the optimal system-level result due to potentially incompatible interactions or unforeseen effects on how the generator utilizes the context provided by the reranker. This highlights the need for evaluating interacting components together as functional 'blocks' within the pipeline, rather than relying solely on optimizing components in isolation. Rigorous testing of potential component block configurations becomes essential.

How to evaluate such component blocks? We can differentiate two major functional blocks where such interactions are common: pre-retrieval and post-retrieval. The evaluation of pre-retrieval component blocks (e.g., involving query rewriters) often relies on observing their impact on subsequent retrieval performance. The primary goal of pre-retrieval steps is typically to enhance retrieval quality, ensuring the retriever finds the necessary documents or chunks for the task. Common pre-retrieval techniques include query routing, query transformation, and query expansion. Additionally, parameters adjusted during the data ingestion stage, such as chunking methods, document

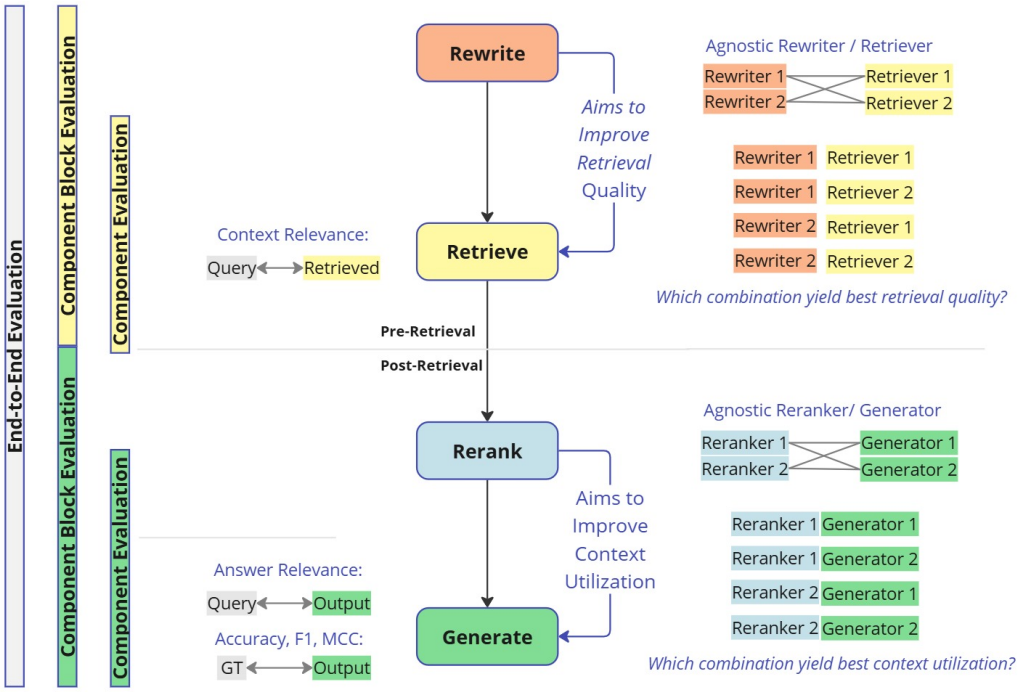

Figure 4.5: Comparison of isolated component evaluation of each component and component block evaluation, where several components with the same goal are evaluated.

selection criteria, and preprocessing steps, also significantly influence subsequent retrieval quality and can be considered part of optimizing the broader pre-retrieval process.

Post-retrieval components, such as rerankers, primarily aim to improve context utilization by the generator. Ideally, the generator receives a concise, curated list of documents tailored to its processing capabilities and the specific query, and rerankers help achieve this by reordering the initially retrieved set. Evaluating the impact of this reranking step is crucial as it directly influences overall system performance. The effectiveness of the reranker block can be assessed using metrics similar to those used for retrieval (e.g., Context Relevance, MAP@K applied to the reranked list) or by its downstream impact on end-to-end performance.

Since many components beyond the core retriever and generator lack direct, intrinsic evaluation metrics, their effectiveness must often be assessed indirectly through replacement experiments. In this approach, different variations of a component or block are tested while keeping the rest of the pipeline

constant, and the results are compared using relevant block-level or end-to-end metrics. For instance, different rewriter approaches (e.g., using various models, prompting techniques, or alternative components) can be evaluated as distinct variations. The RAG system is then evaluated with each variation. The configuration yielding the best performance on relevant metrics (e.g., end-to-end accuracy or retrieval metrics measured after the combined Rewrite-Retrieve steps) is considered superior for that specific pipeline configuration. This comparative evaluation approach necessitates testing multiple RAG variations to enable rigorous assessment. The next section introduces our approach to facilitate such multi-configuration experiments efficiently.

4.3 Fast RAG Development

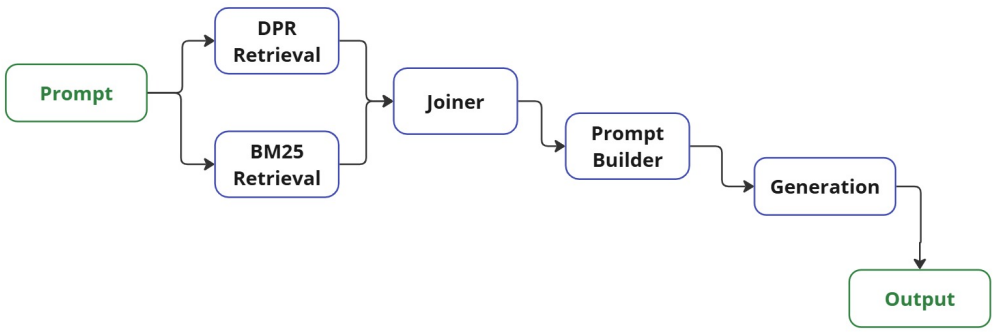

Figure 4.6: A simple Retrieve-Read pipeline with both dense and sparse retrieval, definable via YAML.

RAG systems possess numerous parameters whose optimal settings are often non-obvious. Researchers typically undergo multiple reconfiguration phases until the RAG system achieves satisfactory performance. Intuitively, more iterative refinement cycles can lead to better results. Therefore, enabling short feedback cycles and rapid development is crucial; it allows for the evaluation of numerous configurations and facilitates quick identification and mitigation of performance bottlenecks. This section outlines our approach and considerations for enabling faster RAG development cycles.

Several RAG development tools and frameworks exist. Widely used ones include Llama-Index [51], Langchain [8], and Haystack [60]. All offer comparable functionality for building advanced RAG systems with a modular architecture, as introduced by Gao et al. [21]. Haystack provides custom components that support new technologies and offers functionality for users to define pipelines

via YAML files. This enhances the reconfigurability of these systems, as it involves changing parameters in a single YAML file instead of editing Python files. It also improves the reporting of experimental methodologies: instead of saving a Python script or module for each configuration, only a YAML file describing the tested RAG architecture is stored. An example of this YAML definition can be seen in Figure 4.6 and the YAML code below.

Configuring via YAML files offers additional benefits. First, existing RAG configurations can be copied easily. Second, pipeline definitions can still be created in Python and subsequently saved in the YAML format. Lastly, Haystack is developing a UI [14] for RAG system development, which can be used to create complex systems in a two-dimensional space.

```yaml
components:
  llm:
    init_parameters:
      api_base_url: null
      api_key:
        env_vars: OPENAI_API_KEY
      ...
  prompt_builder: ...
  bm25_retriever: ...
  embedding_retriever: ...
  joiner: ...
  text_embedder: ...
  docs_embedder: ...
  answer_builder: ...
connections:
- receiver: llm.prompt
  sender: prompt_builder.prompt
- ...
```

While copying configuration files and adjusting their parameters is more efficient than handling verbose Python modules, we expanded Haystack's core functionality for saving and loading YAML configurations with matrix notation. This allows specifying different parameters within a list to generate all possible combinations of that configuration. In the example below, four different combinations can be seen: ("gpt-4o-mini", 5), ("o3-mini", 5), ("gpt-4o-mini", 10), and ("o3-mini", 10). This ensures that researchers can test and validate numerous models or parameters and compare their impact on overall performance.

```yaml
components:
  llm:
    init_parameters:
      model: ["gpt-4o-mini", "o3-mini"]
      ...
  retriever:
    init_parameters:
      top-k: [5, 10]
  ...
```

Vector databases can differ in functionalities and performance, especially during ingestion. In this framework, we support three different vector databases for experimentation:

- In-Memory built-in vector database by Haystack
- Chroma vector database [11]
- Qdrant vector database [62]

4.4 Transparency

Following the recommendations of Simon et al. [71] for ensuring transparency in RAG experiments, several practices are crucial. First, transparency requires ensuring that the data used is either publicly available or published alongside the experimental results. While managing data accessibility itself falls outside the direct scope of this framework - typically handled via data repositories or standard version control systems like Git, often hosted on platforms such as GitHub [35] or GitLab [22] - we acknowledge its importance. Data Version Control (DVC) [33] is another popular tool, primarily focused on traditional machine learning experiments. However, its stage-based workflow can introduce overhead that may hinder the rapid iteration cycles common in RAG development. Additionally, its visualization features are often tailored to epoch-based training, which is less relevant for typical RAG evaluation workflows.

Within our recommended workflow, both the framework code and the RAG system architecture should be version controlled (e.g., using Git) with each commit. This practice ensures that each experimental configuration can be precisely restored and transparently reported.

We facilitate this further by enabling the inclusion of metadata parameters directly within the configuration files, a practice we highly recommend for

every configuration phase. Researchers can embed comments, document hypotheses, or record observations within the configuration itself, which can later aid in comparing results across different runs within MLflow. Additionally, this metadata makes the rationale behind specific configuration changes more transparent. The benefits include: first, clarifying the researcher's intent (e.g., targeting retrieval versus context utilization improvements); and second, documenting unsuccessful configurations or hypotheses, which can provide valuable negative results for the wider research community.

```yaml
components:
  ...
connections:
- ...
metadata:
  comment: "This is a comment"
  hypothesis: "This is a hypothesis"
  seed: 42
```

Reproducibility in machine learning often relies on random seeding. This can be complex in RAG systems, as not all components or underlying services may expose seeding capabilities. We define seeds in our validation-test split and emphasize the importance of utilizing seeding wherever component parameters allow.

The transparency measures discussed here - publishing data, versioning code and configurations, embedding metadata, and utilizing seeding - contribute significantly to the reproducibility of RAG experiments conducted using this framework. Our approach primarily assumes a relatively static dataset during an experimental cycle. Handling frequently changing datasets might necessitate integrating tools like DVC, which is currently outside the framework's scope. Besides data stability, full reproducibility depends on the ability to define a seed for all components that include randomness.

4.5 Validity

Internal Validity Evaluating semantic correctness in LLM and RAG system outputs often relies on either LLM-as-a-Judge models or lexical overlap metrics such as BLEU or ROUGE, which compare the generated output tokens against a reference ground truth. Such lexical metrics often fail to capture semantic equivalence between outputs that use different wording. While LLM-as-a-Judge approaches often correlate well with human judgment, they

are susceptible to issues like preference leakage. Research suggests that LLM judges may exhibit biases, potentially favoring outputs from LLMs similar to themselves [47]. This field remains an active area of research. In this framework, our LLM-as-a-Judge metrics (e.g., for Context Relevance) utilize a configurable judge model, defaulting via environment variables to OpenAI's *gpt-4o-mini* [57]. We actively caution users about this potential bias and provide the flexibility to configure a judge model from a different family than the generator LLM being evaluated, mitigating potential self-preference issues.

External Validity Our framework addresses a key aspect of external validity through its mandatory use of a validation-test split. As described previously (Section 4.1), the evaluation dataset is split, and all system tuning and reconfiguration phases are performed using *only* the validation set. This holds true regardless of whether these phases involve simple parameter tuning or training custom components (like fine-tuning a retriever or generator). The held-out test dataset is used only once at the final stage to estimate the generalization performance of the chosen configurations. Within MLflow, users can easily compare the metrics obtained on the validation set during development against the final metrics calculated on the test set for each experimental run. Significant discrepancies between validation and test performance highlight configurations that may have overfitted to the validation set, failing to generalize well.

Figure 4.7: Comparison of reconfiguration between RAGs and LLMs - both relying on tuning parameters and test data.

4.6 User Interface

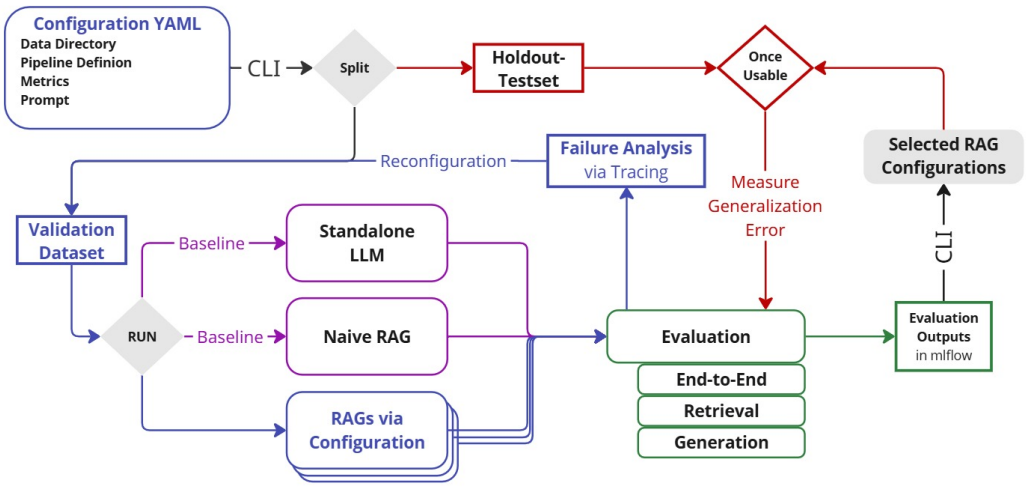

Figure 4.8: Framework: First, the evaluation data is split into validation and hold-out test datasets. Then, the validation data is used to evaluate a configured RAG system and compare it against baselines. After failure analysis, several reconfiguration phases can occur. Finally, the test dataset is used to test all configurations for potential generalization error.

Leveraging the previously discussed Haystack functionalities (like YAML pipelines), we have developed a Command-Line Interface (CLI)-based evaluation framework. The complete workflow orchestrated by the framework is depicted in Figure 4.8. Given paths to an evaluation dataset, pipeline configurations, and metric definitions, the framework first splits the dataset into validation and test sets based on the specified *test_size* parameter. Subsequently, it uses the validation split to conduct the evaluations.

The framework loads the necessary pipeline components and data and begins by evaluating the baselines: first, a standalone LLM (to assess the value of retrieval itself), and second, after ingesting the data into the specified vector database, a naive BM25-based RAG (Retrieve-Read architecture) baseline (to establish a simple RAG performance level). Following the baseline evaluations, the framework executes and evaluates the RAG pipeline(s) defined in the user-provided YAML configuration file(s). Each evaluated configuration (baselines and user-defined) is assessed using both end-to-end and component-level metrics (as configured). Results (metrics and parameters) are logged to MLflow for visualization and comparison (Figure 4.9), while detailed execution traces are sent to Langfuse for inspection (Figure 4.10). Based on the results and

trace analysis, researchers can reconfigure the pipeline YAML and repeat the evaluation process on the validation set.

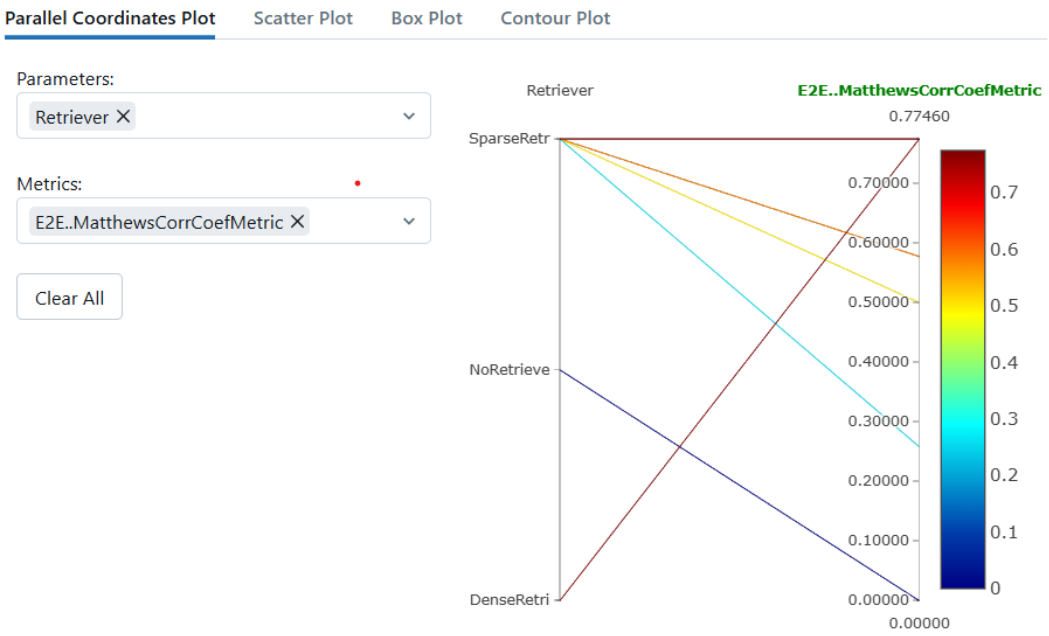

Figure 4.9: Example of an MLflow run view in our framework. Hypothesis, comments, etc., can be defined via metadata within configuration files. These annotations facilitate comparison and interpretation of results.

The framework's CLI, built using Typer, provides commands to streamline the evaluation workflow. The two primary commands are: `run-evaluations`, which executes the main evaluation loop on the validation set (running baselines and specified YAML configurations); and `test-generalization`, which runs specified configurations against the held-out test set to assess generalization. Both commands log results and parameters to MLflow and traces to Langfuse. This integration supports efficient, reproducible, and transparent RAG development. The framework is publicly available on GitHub [2].

4.7 Limitations

The current implementation of this framework primarily focuses on evaluating RAG systems for classification tasks, with built-in support for relevant metrics. However, its design allows for expansion via custom metric implementations, enabling users, in principle, to adapt it for evaluating generative tasks by defining appropriate metric classes.

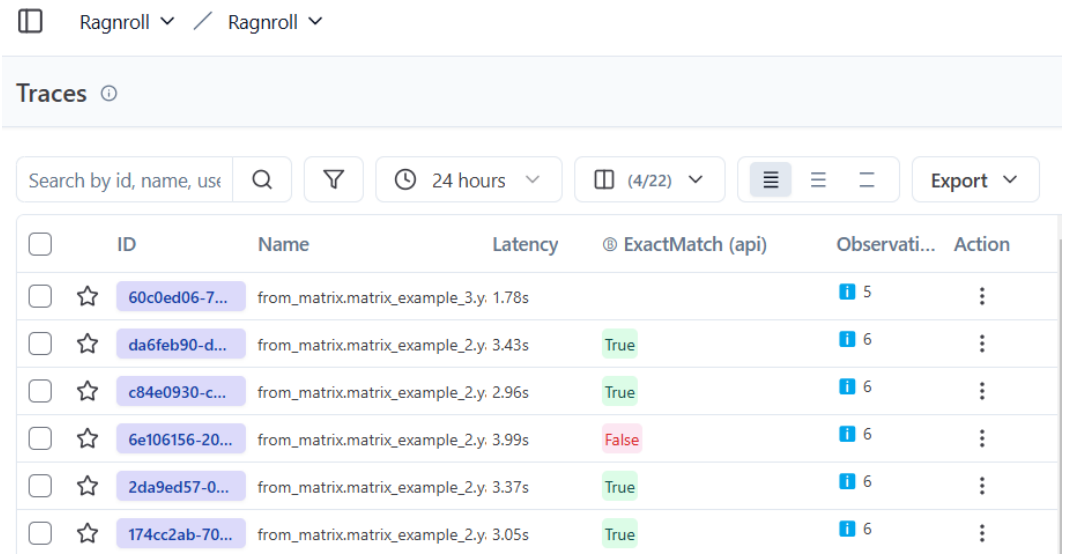

Figure 4.10: Example Langfuse trace view. Traces log input, output, and intermediate steps for each query, tagged here with success status to aid failure analysis.

We implemented the framework to be as modular as possible so that any conceivable architecture with all customized components can be evaluated. However, we cannot guarantee that every architecture is possible, and supporting agentic workflows might require further work.

Chapter 5

RAG Experimentation on Configuration Validation

5.1 Introduction

Modern software systems involve intricate changes that extend far beyond source code modifications. A significant portion of system behavior is governed by configuration files, encompassing deployment scripts, service settings, infrastructure definitions, and more. Ensuring the correctness of these configurations is critical, yet challenging. Although software vendors often provide extensive manuals to guide administrators, the length and complexity of these documents can be prohibitive, frequently leading practitioners to resort to guesswork when configuring systems [82]. This challenge is compounded by the rapid pace of technological evolution, which necessitates constant updates to configuration schemes.

The consequences of misconfiguration can be severe, significantly contributing to production bugs and system failures, as highlighted by large-scale studies [73]. Traditionally, validating configurations has relied on methods such as static analysis, integration testing, and manual review [49]. While valuable, these approaches often struggle to keep pace with the complexity and dynamism of modern software environments. Furthermore, the scale at which validation is required can be immense; Facebook, for example, reported executing trillions of configuration checks daily [73]. This necessitates solutions that are not only accurate but also highly performant and resource-efficient, implying that evaluation solely based on metrics like precision or recall is insufficient. Consequently, there is a pressing need for more automated, reliable, and scalable validation techniques.

Retrieval-Augmented Generation (RAG) systems offer a promising approach to address this gap. By combining the knowledge retrieval capabilities of search

systems with the reasoning power of Large Language Models (LLMs), RAG can potentially interpret complex technical documentation (like configuration manuals) and apply this understanding to validate specific configurations against best practices or requirements.

This chapter aims to demonstrate the practical application of the RAGnRoll evaluation framework, detailed in Chapter 4, to the specific problem of configuration validation. We will leverage this framework to systematically evaluate the performance of various RAG configurations against established baselines using an extended version of the CTest dataset [49], which includes synthetically generated examples. While acknowledging that configuration validation is a multifaceted problem involving aspects like inter-configuration dependencies, this experiment focuses specifically on validating individual configuration settings based on documented guidelines.

The remainder of this chapter is structured as follows: Section 5.2 discusses related work in applying RAG to software engineering tasks, particularly configuration validation. Section 5.3 details the experiment design, including the dataset, baselines, RAG configurations, evaluation metrics, and tools used, adhering to the methodology outlined in Chapter 4. Section 5.3.2 presents and analyzes the results obtained on the validation and test sets, including end-to-end performance, component analysis, and failure analysis. Section 5.4 discusses the overall findings and their implications. Finally, Section 5.6 summarizes the key outcomes of the experiment.

5.2 Related Works

Several approaches have been developed for validating software configurations.

Specification-based frameworks, such as ConfValley [30], function by having engineers define validation rules explicitly, often in a declarative manner. Configuration Testing validates configuration values by executing the software components that use these values and observing the resulting program behavior [72]. This method is used to detect the dynamic effects of configuration settings and identify errors.

Lian et al. [49] introduced Ciri, a method that uses Large Language Models for configuration validation. Ciri employs prompt engineering and few-shot learning, providing the LLM with examples of both valid and invalid configurations alongside relevant documentation snippets to identify misconfigurations. This work applies Retrieval-Augmented Generation to the configuration validation task presented by Lian et al. [49]. We utilize the RAGnRoll evaluation framework, detailed in Chapter 4, to systematically assess and reconfigure different RAG systems. The aim is to optimize their performance for this specific

task through iterative refinement.

5.3 Experiment Design and Execution

The following sections detail the design and execution of our configuration validation experiment. This experiment adheres strictly to the methodology established by the RAGnRoll framework presented in Chapter 4. The core principle is to split the evaluation dataset into validation and test sets. All system configuration, evaluation, and iterative refinement are performed exclusively on the validation set. Finally, generalization is assessed on the held-out test set. Specifically, we employed a 80/20 validation-test split for the extended CTest dataset to ensure a sufficiently large test set for estimating generalization error (as discussed in Section 4.1).

Our experimental workflow begins by establishing performance baselines using a standalone LLM and a naive RAG system (Section 4.2.1). We then evaluate an initial RAG configuration. Subsequent steps involve analyzing performance bottlenecks and failures using the framework's end-to-end and component-level metrics, alongside detailed trace analysis facilitated by MLflow and Langfuse (Section 4.6). Insights from this analysis guide iterative reconfiguration cycles aimed at improving performance on the validation set.

Ciri avoids a retrieval architecture by randomly selecting valid configurations and ingesting them into the evaluation prompt. Then, three different generators are used in parallel to obtain a majority vote on whether the provided configuration is valid. The drawn architecture is visualized in the appendix. A key difference in our evaluation approach compared to the original Ciri experiment [49] lies in handling output formatting. While Ciri reran queries until a correctly formatted answer was obtained, our framework employs a strict format check; if the generator's output does not conform to the expected format (such as providing a clear "valid" or "invalid" classification with reasoning), the response is marked as incorrect. This reflects a scenario where automated post-processing requires predictable output.

5.3.1 Experimental Setup

The experiments were conducted using the computational resources specified below. A dedicated server hosted by Hetzner served as the primary environment for running the evaluation framework, encompassing data processing, baseline evaluations, and RAG pipeline executions that involved API-based LLMs or CPU-based components. We selected both closed and open-source models and utilized

OpenRouter.ai [58] for all API requests. We opted not to self-host models, as this could skew system metrics such as latency.

We chose the CCX23 dedicated CPU server[23] for all our tests

- VCPU: 4
- RAM: 16 GB
- SSD: 160 GB

We also used runpod.io[66] and deployed a RTX 4090 with 24GB VRAM for our experiments with via vLLM [45] deployed embedding models.

Dataset We used the CTest dataset, prepared by the team behind Ciri [49][83]. The data consists of real-world misconfiguration scenarios, augmented with synthetic data. We had 907 test cases in total. The distribution per system is shown in the following table.

Technology	Number of Files	Version
alluxio	111	2.5.0
django	36	4.0.0
etcd	64	3.5.0
hbase	107	2.2.7
hcommon	138	3.3.0
hdfs	149	3.3.0
postgresql	62	13.0
redis	88	7.0.0
yarn	80	3.3.0
zookeeper	72	3.7.0

Table 5.1: Evaluation Dataset: Number of configuration files and versions per system.

We then defined the validation-test split. We chose a test size of 20%, resulting in 725 validation data points and 182 test data points.

Ingesting data into our vector database required scraping official documentation for each specific version. However, we could not find documentation or a manual for this specific Redis version, nor for HBase, HCommon, and HDFS. Table 5.2 shows the documentation sources we scraped.

Data ingestion was performed ad hoc before each experiment to ensure comparable results for ingestion times.

Technology (Version)	Documentation Page
alluxio (2.5.0)	https://docs.alluxio.io/os/javadoc/2.5/
django (4.0.0)	https://docs.djangoproject.com/en/4.0/
etcd (3.5.0)	https://etcd.io/docs/v3.5/
hadoop-	https://hadoop.apache.org/docs/stable/ hadoop-project-dist/
hbase (2.2.7)	hadoop-common/
hcommon (3.3.0)	hadoop-common/
hdfs (3.3.0)	hadoop-hdfs/
postgresql (13.0)	https://www.postgresql.org/docs/13/
redis (7.0.0)	https://redis.io/docs/latest/commands/
yarn (3.3.0)	https://hadoop.apache.org/docs/r3.3.0/
zookeeper (3.7.0)	https://zookeeper.apache.org/doc/r3.7.0 /apidocs/zookeeper-server/

Table 5.2: Technology and Documentation Links: Starting at those documentation pages as base URLs, we scraped every linked page with the same base URL as a prefix recursively.

5.3.2 Reconfiguration Phases

Initial Configuration and Baselines We used several baselines for comparison. Initially, we ran our experiment with the baselines defined in Section 4: a standalone LLM and a predefined minimal RAG system with BM25 sparse retrieval. For both baselines, we used OpenAI's GPT-4o-mini [57]. The standalone system consists of only a generator and an answer builder for extracting "valid" or "invalid" from the generated output. The second baseline adds a retriever immediately before the generator and lists the top-10 documents without any pre- or postprocessing. These basic architectures allow us to determine if complex adjustments improve validator quality.

We also rebuilt the Ciri system within our Haystack RAG architecture and used it as a baseline. Since we only used our framework for running evaluations, there are some minor methodological differences due to implementation limitations. We did not analyze results per system, meaning we do not have specific knowledge of whether our system performs better for Django or Alluxio systems. Although we could have run separate experiments for each system, we opted against this for better comparability of the final results and to reduce experimental costs. We also only evaluated file-level validation. Ciri used few-shot examples with a number of valid and invalid configurations. We did not vary the number of valid or invalid shots; instead, we chose the best-performing configuration with one valid and three invalid shots. Lastly, Ciri used several

language models for its experiments. We were limited to using *gpt-3.5-turbo-0125*[57]. Other models were either unavailable because the used version was outdated, continuously trained, or too expensive for our experimental budget. However, we achieved an F1-Score of *0.680*, which is close to the Ciri-reported F1-Score of *0.720*. The pipeline figures can be found in the appendix.

Initial RAG Configuration First, we tested different standalone LLMs to measure their performance without a complex system architecture. For our evaluation, we chose Qwen's *QwQ-32B* [75] and *Qwen-2.5-coder-32b-instruct* [32][85][86], OpenAI's versioned *o1-mini*, an earlier version of *gpt-4o-mini* (*gpt-4o-mini-2024-08-06*) and an up-to-date version of *gpt-4o-mini* for comparison. We also tested OpenAI's web-search version of *gpt-4o-mini*, which appears to be an out-of-the-box RAG system [57]. Lastly, we added DeepSeek's most recent model, *V3* [13], to the list.

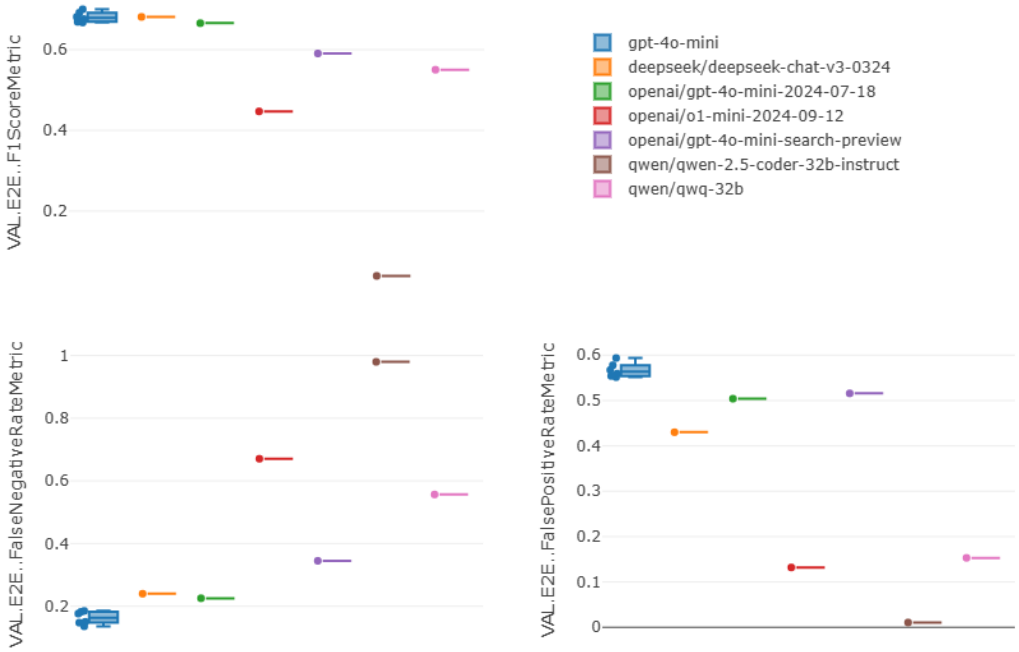

Figure 5.1: (Upper) F1-Scores for standalone Large Language Models on the configuration validation task. (Lower) False-Negative Rate (Left) and False-Positive Rate (Right) for the standalone LLMs on the configuration validation task. Metric names are concatenated by *VAL* refers to validation dataset, *E2E* refers to end-to-end metric and the metric name.

Results can be seen in Figure 5.1. The most recent *GPT-4o-mini* model performs best for F1-Score. The evaluated models differ more strongly on

False-Positives Rate and False-Negative Rate than on F1-Scores. *GPT-4o-mini* has the lowest False-Negative-Rate and the highest False-Positive-Rate. Since positives represent valid configurations, *GPT-4o-mini* classifies invalid configurations as valid more often, and conversely, classifies valid configurations as invalid less frequently. The older version of *GPT-4o-mini* from 2024 and the web-search model have a slightly worse F1-Score, but more balanced FNR and FPR metrics. Even without few-shot learning as done by the Ciri team, the most recent version of *gpt-4o-mini* did score a slightly higher average F1-Score (0.681) than the configuration from the Ciri team with *gpt-3.5-turbo-0125* (0.680). In contrast to other models, we ran the most recent version of *gpt-4o-mini* more frequently as a baseline in later experiment runs. The variance in F1-Scores in these runs is relatively small (1.41×10^{-5}).

Next, beyond the basic BM25 retriever (predefined as a baseline), we evaluated additional retrieval configurations. First, we tested a configuration employing only a dense retriever with OpenAI's *text-embedding-3-small* [57]. Second, we implemented a hybrid retrieval configuration that combined the results from the BM25 retriever and the aforementioned dense retriever, using three different weighting schemes for their respective scores: (1:2), (1:1), and (2:1), where the ratios represent (BM25 score : Dense score). We evaluated the performance of these configurations with both the *gpt-4o-mini-2024-08-06* and the most recent *gpt-4o-mini* model. The results are shown in Figure 5.2 and indicate that the selected model has more impact on overall performance than the retrieval technique in this experiment. The retrieval metrics are presented in Table 5.3. Overall, the sparse retriever performed better than the dense retriever, although both techniques performed poorly on retrieval quality. The weight distribution for the hybrid retriever had only a small impact on performance, with F1-Scores ranging from *0.567* to *0.582*.

Metric	**BM25**	**Dense**
ContextRelevance	0.396	0.176
MAP@K	0.096	0.030

Table 5.3: Retrieval metrics comparison between BM25 and Dense retrieval

Following this phase, we attempted to improve the resource quality of the data. The scraped documentation HTML files were extensively cleaned by removing numerous elements irrelevant to configuration parameters, including structural page elements (headers, footers, navigation bars, sidebars), presentational tags (styles, SVGs, images), and embedded code (scripts). We removed all these elements and kept only the relevant ones containing text or metadata. The cleanup statistics are shown in Table 5.4. However, the retrieval quality for sparse retrievers unexpectedly decreased from *0.693* to

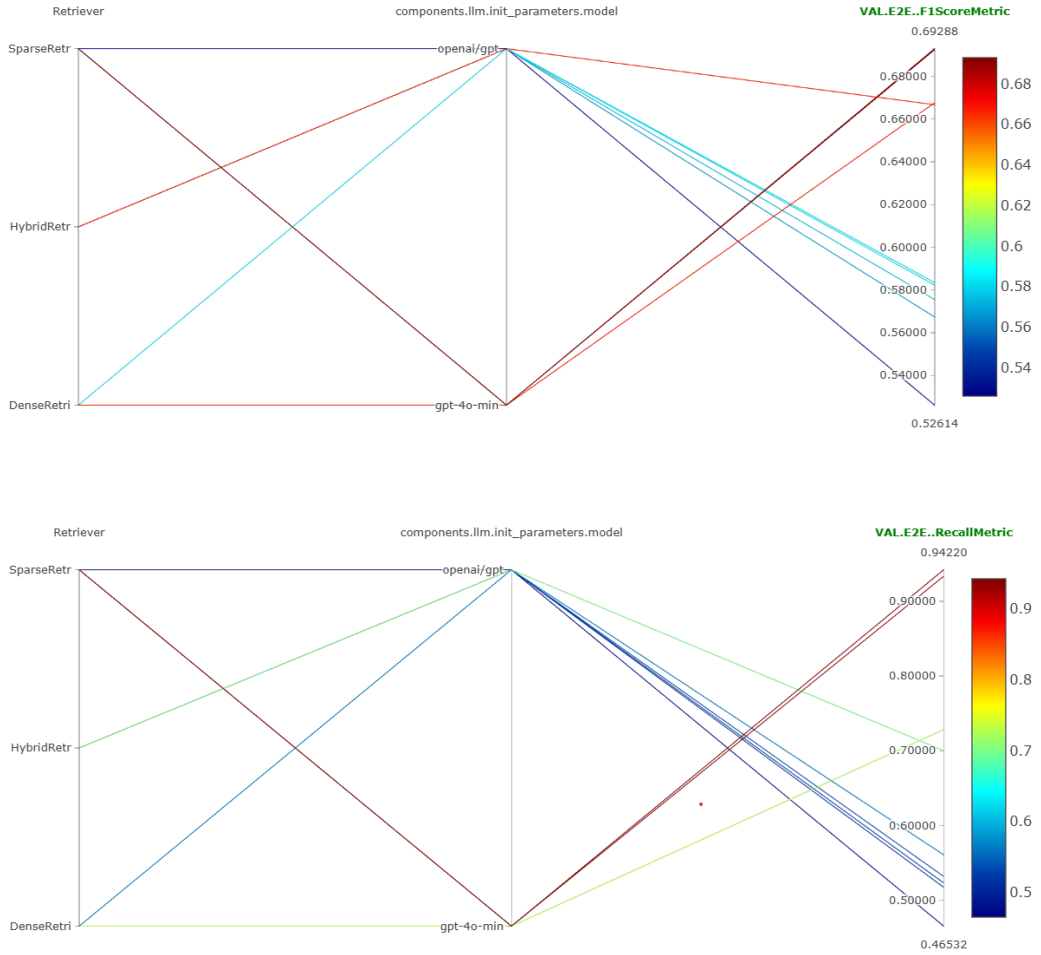

Figure 5.2: Experiments with different retrieval techniques and models. In both figures the upper model is the *gpt-4o-mini-2024-08-06* model and the lower model is the most recent *gpt-4o-mini*. The upper figure shows the F1-score and the lower one shows the recall score. Metric names are concatenated by *VAL* refers to validation dataset, *E2E* refers to end-to-end metric and the metric name.

0.567.

As the results were not improving, we felt forced to change more parameters within a single configuration step. Therefore, for the dense retriever, in addition to using the improved quality data, we also tried a different embedding model: *infly/inf-retriever-v1-1.5b* [87]. We selected this model because it is small enough for self-hosting and scores best in its size class in the MTEB [56][31] "Code" category. We also applied a 4R-architecture with a rewriter that transforms the configuration question to be validated into hypothetical

System	Files	Original Size	New Size	Reduction
alluxio	230	20.64MB	19.58MB	5.16%
django	634	37.10MB	22.05MB	40.57%
etcd	113	19.27MB	1.86MB	90.33%
hbase	440	26.42MB	18.30MB	30.74%
hdfs	108	2.83MB	2.09MB	26.18%
postgresql	1137	26.50MB	22.94MB	13.43%
redis	515	93.69MB	8.33MB	91.11%
yarn	563	40.93MB	27.43MB	32.99%
zookeeper	35	2.29MB	2.12MB	7.45%
TOTAL	**3775**	**269.68MB**	**124.70MB**	**37.55%**

Table 5.4: HTML Documentation Cleanup Statistics

documentation pages, allowing the embedding retriever, in theory, to map these hypothetical documents to real ones in the embedding space. Lastly, we added a Lost-in-the-Middle reranker [53]. The results improved greatly, achieving an F1-Score of *0.6825*, which exceeded the results we measured for Ciri.

Lastly, we wanted to update Ciri's results with more recent state-of-the-art models. Therefore, we tested the Ciri configuration with several models: *gpt-4o-mini-2024-08-06* and the most recent version of *gpt-4o-mini* [57], *gemini-2.5-flash* [59], *deepseek-chat-v3-0324* [13], and the models from Alibaba released in April 2025, *qwen3-32b* and *qwen3-235b-a22b* [74]. We evaluated these models in two scenarios: one without explicit reasoning before answering the question and one with reasoning before answering the question.

Model	Without Reasoning			With Reasoning		
	F1	**P**	**R**	**F1**	**P**	**R**
gpt-4o-mini	0.688	0.785	0.613	0.508	0.796	0.373
gpt-4o-mini-2024-08-06	0.699	0.784	0.630	0.501	0.776	0.370
gemini-2.5-flash	0.747	0.806	0.697	0.456	0.829	0.315
deepseek-chat-v3-0324	**0.776**	0.794	**0.759**	0.666	0.790	0.575
qwen3-32b	0.609	0.810	0.488	0.603	**0.841**	0.470
qwen3-235b-a22b[1]	-	-	-	-	-	-

Table 5.5: Comparison of model performance (F1-Score, Precision (P), and Recall (R)) with and without explicit reasoning. [1]Results for Qwen-3's Mixture-of-Experts model *qwen3-235b-a22b* are incomplete due to errors on many queries; they are included for transparency

Reasoning generally decreased the F1-score and recall across the tested

models. The impact on precision scores was more varied: three models (*gpt-4o-mini*, *gemini-2.5-flash*, and *qwen3-32b*) showed an increase in precision with reasoning, while two (*gpt-4o-mini-2024-08-06* and *deepseek-chat-v3-0324*) exhibited a slight decrease. This finding indicates that while prompting for reasoning consistently lowered recall (identifying fewer of all actual valid configurations), its effect on the precision of those classifications (the proportion of 'valid' classifications that were correct) was model-dependent.

The highest F1-Score (0.776) was achieved by the *deepseek-chat-v3-0324* model within the Ciri configuration. Overall, we achieved good, though not matching, results (0.693) with a sparse configuration with the most recent version of *gpt-4o-mini*. We also came close to this result with a dense 4R-architecture and the *infly/inf-retriever-v1-1.5b* embedder (0.683).

5.3.3 Generalization Test

In this section, we address the generalization error for our experiments. Theoretically, generalization error occurs when parameters are tuned to best fit the evaluation data rather than the underlying problem. During our reconfiguration phases, we had limited success with RAG configurations. Typically, one would identify an effective combination of resources and configuration to address the problem, then fine-tune parameters to achieve the best-performing setup, which would subsequently be tested for generalization error. However, we did not achieve RAG configurations successful enough to warrant such a generalization test focused on potential overfitting from our tuning.

Instead, we decided to perform tests on the held-out test dataset using the best-performing models from Table 5.5 within the Ciri few-shot configuration. While this does not primarily test for generalization error stemming from our own parameter fine-tuning (as these specific model configurations were not iteratively tuned by us), it provides a valuable estimation of their performance variance on unseen data. This helps assess the consistency of these standalone LLM setups in this task. We selected *deepseek-chat-v3-0324*, *gemini-2.5-flash*, and the original Ciri configuration with *gpt-3.5-turbo* for this purpose.

Model	**Validation Set**			**Test Set**		
	F1	**P**	**R**	**F1**	**P**	**R**
gpt-3.5-turbo	0.680	0.735	0.633	0.650	0.663	0.639
gemini-2.5-flash	0.747	0.806	0.697	0.735	0.735	0.735
deepseek-chat-v3-0324	**0.776**	0.794	**0.759**	0.763	0.733	0.795

Table 5.6: Comparison of model performance between validation and test sets with F1-Score, precision (P) and recall (R).

5.3.4 System Metrics

First, we present the different indexing times for the two embedding models that we tested. Figure 5.3 shows both the OpenAI embedding via API and the vLLM embedding via a runpod.io deployment. OpenAI's embedding API required 3.5 times more time for indexing.

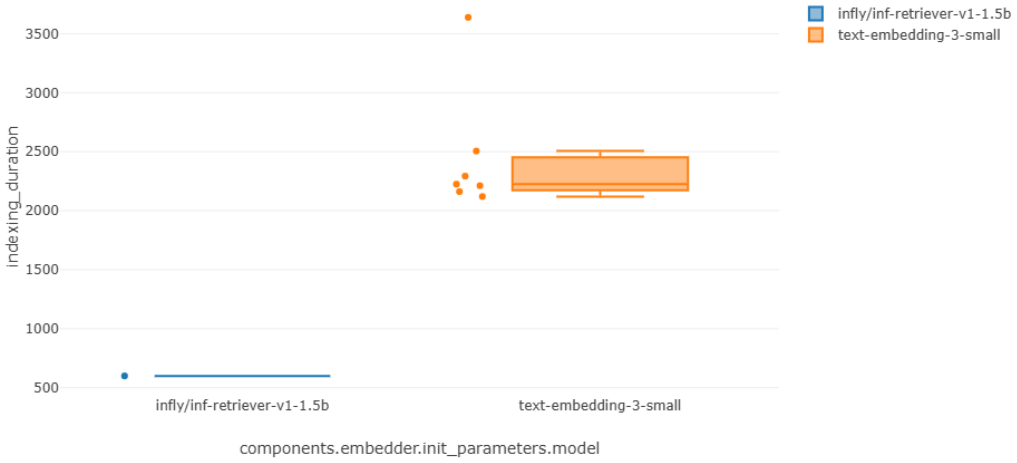

Figure 5.3: Indexing duration comparison between the two tested embedding models: *openai/text-embedding-small-3* and *infly/inf-retriever-v1-1.5b*.

We also measured evaluation time to assess the usability implications of different RAG systems. The objective was to determine whether the performance boost offered by a RAG system justifies the potential drawback of longer computation times compared to simpler approaches. Since the best-performing configurations identified in our experiments were based on standalone LLMs using the Ciri few-shot approach (rather than more complex RAG pipelines), these top-performing systems were also inherently among the fastest in terms of computation time. As clearly shown in Figure 5.4, using a retriever of any type substantially increases computation times compared to the standalone LLM baseline (excluding data ingestion time).

We also measured system memory usage (e.g., peak RAM) to estimate the operational cost associated with deploying different configurations. Results are shown in Figure 5.5. Configurations involving hybrid and dense retrievers required the most RAM.

Lastly, we consider the costs associated with the models and embeddings. It was not possible to gather exact per-token costs paid through OpenRouter due to reporting incompatibilities with Haystack. However, all models processed a similar volume of input tokens, which was the primary driver of API costs. The

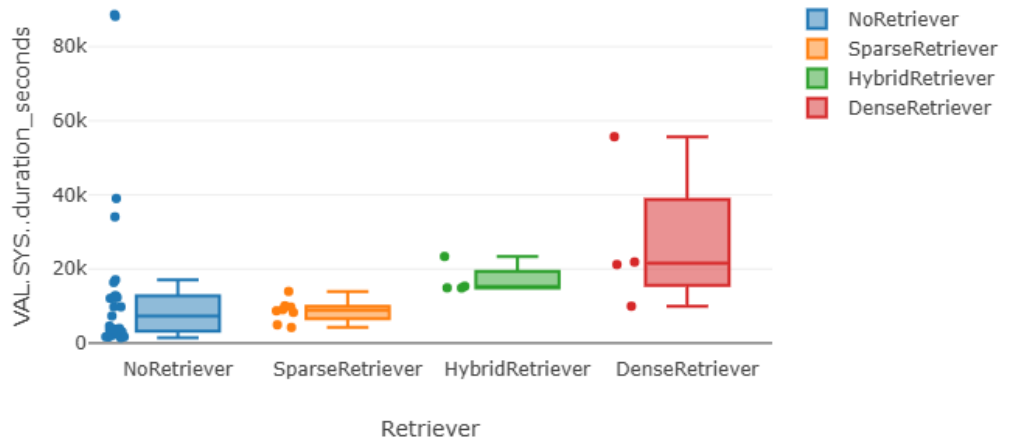

Figure 5.4: Evaluation duration comparison between different retriever types (Sparse, Dense, Hybrid) and the standalone LLM baseline (No Retriever). Metric names are concatenated by *VAL* refers to validation dataset, *SYS* refers to system metric and the metric name.

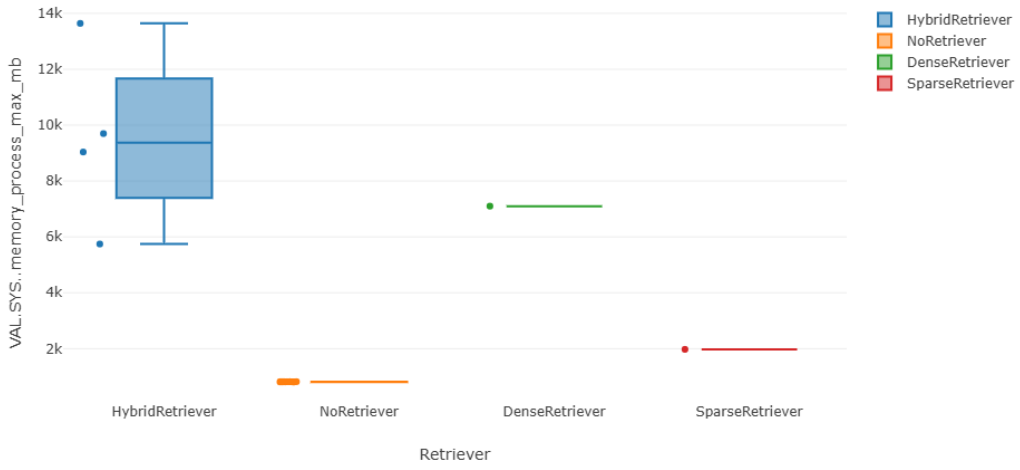

Figure 5.5: Comparison of maximum memory usage for different retriever types (Sparse, Dense, Hybrid) and the standalone LLM baseline (No Retriever). Metric names are concatenated by *VAL* refers to validation dataset, *SYS* refers to system metric and the metric name.

output consisted only of the classification and a brief justification. Therefore, Table 5.7 lists the advertised costs per million tokens for the evaluated models.

Estimating the cost of embedding is more complex. First, consider the Ope-

Table 5.7: Cost comparison of different models per million tokens, grouped by provider

Model Name	Input ($/M)	Output ($/M)
OpenAI		
openai/gpt-3.5-turbo-0125	0.50	1.50
openai/gpt-4o-mini	0.15	0.60
openai/gpt-4o-mini-2024-07-18	0.15	0.60
openai/gpt-4o-mini-search-preview	0.15	0.60
openai/o1-mini-2024-09-12	1.10	4.40
Qwen		
qwen/qwen3-235b-a22b	0.10	0.10
qwen/qwen3-32b	0.10	0.30
qwen/qwen-2.5-coder-32b-instruct	0.06	0.18
qwen/qwq-32b	0.15	0.20
Google		
google/gemini-2.5-flash-preview	0.15	0.60
DeepSeek		
deepseek/deepseek-chat-v3-0324	0.27	1.10

nAI API costs for *text-embedding-small-3*: one run on the validation dataset (725 samples) processed approximately 9 million tokens (including ingestion and query embeddings). At a list price of $0.02 per million tokens, this resulted in a cost of approximately $0.18 for the run. For the self-hosted vLLM embedding, we consider the compute time. The initial indexing took approximately 598 seconds. At $0.69 per hour for the runpod.io instance, this equates to approximately $0.11 for ingestion. However, unlike token-based pricing where indexing dominates, for time-based instance pricing, the total experiment duration is the primary cost driver, as the instance must remain active to embed each query prior to retrieval. Since the relevant experiment run took 6.1 hours, the estimated total cost for using the vLLM embedding was $4.21.

5.4 Discussion

We used several standalone models with the initial prompt design. The model specifically designed for coding, *Qwen-2.5-coder-32b-instruct*, performed poorly. We observed that more recent models, such as the most recent version of gpt-4o-mini, Google's *gemini-2.5-flash*, or DeepSeek's *deepseek-chat-v3-0324*, had higher F1-scores than older models like Qwen's QwQ-32B. Reasoning did not improve results in our scenario. This might be attributable to the fact that

we did not use the models' native reasoning capabilities but rather prompted for artificial reasoning before the final answer. Error analysis showed that reasoning led to formatting errors and poor recall scores. However, as discussed previously (Section 5.3.2), the impact on precision scores was varied.

A low recall score for reasoning suggests that LLMs tend to classify valid configurations as invalid more often than without reasoning. However, as noted, the effect on precision was inconsistent across models. Models that perform poorly tend to have a high False-Negative Rate, as shown in Figure 5.6. The models exhibited greater differences in the False-Negative Rate than in the False-Positive Rate metric. This results in an F1-score that depends more heavily on the ability of the model to classify valid configurations more accurately. All tested models were more reliable at detecting invalid configurations (higher true negative rates, implied by lower FPR variance compared to FNR variance), but the highest-performing models were distinguished by their ability to classify valid configurations more reliably (lower FNR).

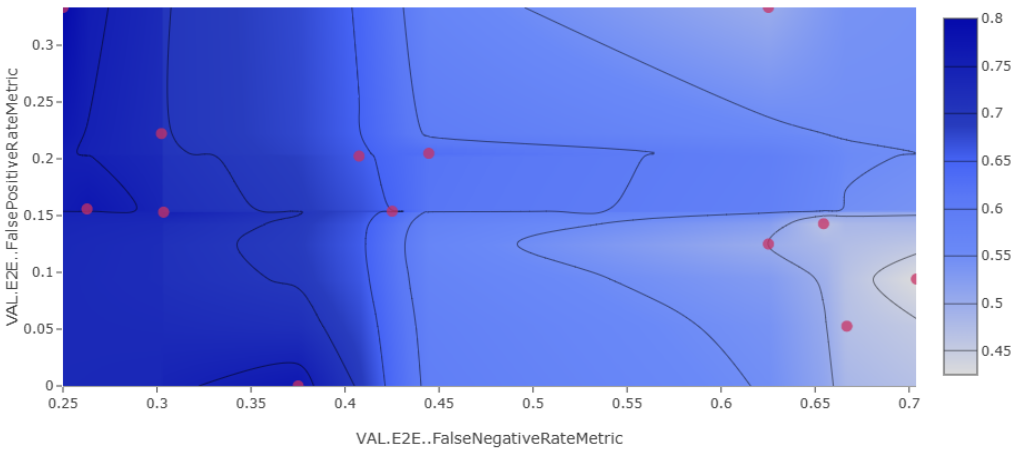

Figure 5.6: Contour plot of FNR and FPR vs F1-Score. The darker the blue, the higher the F1-score. Metric names are concatenated by *VAL* refers to validation dataset, *E2E* refers to end-to-end metric and the metric name.

We assume that if LLMs lack sufficient knowledge about particular configuration parameters, they cannot validate them. Therefore, models whose training data did not include these specific configurations may tend to invalidate them, as they are unaware of the existence of such parameters.

Next, we discuss the results of our RAG architectures. We had several configuration phases that included the following adaptations:

- improving corpus data quality,

- changing retrieval types and their parameters,
- changing generation model,
- changing embedding model.

The bottleneck in the pipeline appeared to be retrieval. In one of our later runs, we were able to improve our results with dense retrieval by using an embedder that is particularly good at coding and an advanced 4R architecture. However, this improvement was not sufficient to surpass the performance of newer standalone models (using the Ciri configuration).

Resource preparation and data preprocessing are more valuable than blindly selecting numerous models or system configurations. The results also showed that data cleanup for the corpus can lead to a decrease in retrieval quality for sparse methods and, simultaneously, an increase for dense methods. However, it should be noted that this observation is partially qualified, as our dense retrieval experiments involved multiple simultaneous changes beyond just data cleanup, including a different embedding model. Improving retrieval quality is the most difficult part of enhancing configuration validation. Developing novel retrieval techniques or training a custom retrieval model are promising solutions to address the poor retrieval quality observed in configuration validation, but these approaches were outside the scope of our experiments. With sufficient monetary resources, one could test many different retrieval configurations with ease. However, our results suggest that small, specialized open-source embedding models (like *infly/inf-retriever-v1-1.5b*) can be more suitable for this problem and also exhibit better performance in terms of ingestion duration compared to the general-purpose proprietary embedding model tested (*text-embedding-3-small*).

Even ignoring ingestion time, systems incorporating a retriever consistently required longer computation times and more RAM during runtime compared to the standalone baseline. Additionally, computation times were also model-dependent, as indicated by outliers for certain configurations in Figure 5.4.

5.5 Threats to Validity

This section discusses potential threats to the validity of the experimental findings presented in this chapter.

Internal Validity Internal validity refers to the extent to which the observed outcomes can be attributed to the experimental manipulations rather than extraneous factors.

- **Baseline Reimplementation:** While efforts were made to faithfully replicate the Ciri system as a baseline, minor differences in our Haystack-based implementation compared to the original Ciri architecture could exist. These might subtly influence direct performance comparisons.
- **API Reliability and Model Stability:** Reliance on external LLM APIs introduces a dependency on their uptime and consistent behavior. API errors or undocumented changes in underlying unversioned models (such as the most recent versions of *gpt-4o-mini* or *gemini-2.5-flash*, used to test contemporary performance) could lead to failed queries or performance variations. While some of these failures might average out across many queries, they could also reflect issues with specific models or providers beyond our direct control.
- **System-Level Aggregation:** The decision to evaluate performance across all software systems in the CTest dataset, rather than per individual system (e.g., Django vs. Alluxio), was made for comparability and cost-efficiency. However, this aggregation might mask varying performance characteristics of the RAG configurations on specific systems.
- **LLM-as-a-Judge for Retrieval Metrics:** While we mitigated preference leakage by ensuring that the LLM judge (e.g., *gpt-4o-mini*) was from a different family than the model generating the content being assessed for retrieval metrics where possible, the inherent subjectivity and potential biases of LLM judges remain an area of active research and could influence metrics like Context Relevance.

External Validity External validity concerns the generalizability of the experimental findings to other contexts.

- **Corpus Specificity:** The document corpus was constructed specifically for the software systems and versions present in the CTest dataset. Therefore, the performance results, particularly for RAG configurations reliant on this corpus, may not directly generalize to newer software versions, different software systems, or entirely different documentation sets.
- **Task Scope:** This experiment focused on file-level configuration validation. The findings may not be directly applicable to parameter-level validation, inter-configuration dependency checking, or other types of software engineering tasks that might require different RAG strategies or knowledge representations.

- **Dataset Representativeness:** While CTest includes real-world and synthetic misconfigurations, the specific distribution and nature of these examples might not encompass all possible configuration errors or documentation complexities found in other operational environments.

Construct Validity Construct validity relates to how well the operationalizations (e.g., metrics, experimental setup) measure the intended theoretical constructs.

- **Metrics for Validation Quality:** Standard classification metrics (F1-score, precision, recall) were used. While informative, they might not fully capture nuanced aspects of validation quality, such as the interpretability or actionability of the reasoning provided alongside a "valid" or "invalid" judgment. We used the F1-Score for all conclusions for better comparability with the Ciri paper.

- **Implementation Integrity of the Evaluation Framework:** The RAGnRoll evaluation framework was developed from scratch for this thesis. During the experimental phase, isolated software bugs within the framework were discovered. For instance, some initial experiment runs were affected by premature terminations, which could have skewed end-to-end performance scores. Another identified issue involved inaccuracies in the reporting of retrieval metrics in early iterations. To mitigate these threats, any experiments where end-to-end metrics were potentially compromised by such bugs were rigorously repeated. While retrieval metrics are primarily used for fine-tuning RAG configurations rather than for final comparative judgments, discrepancies in their early reporting were noted. It is important to state that all metrics presented and discussed in this chapter have been verified for correctness and are based on stable, bug-free versions of the reporting mechanisms.

- **Evolving Reporting Detail:** Throughout the course of the experiments, the necessity to log additional contextual parameters for enhanced traceability and to ensure the unambiguous association of results with specific experimental conditions became apparent. For example, parameters such as the corpus data path and the evaluation data path were explicitly added to the run logs when these paths were first varied. Similarly, the Git commit hash of the codebase was incorporated into the logging process at a later stage to precisely track code versions used for each experiment. Care was taken to ensure that these evolving reporting practices did not compromise the differentiability of results from distinct experimental setups. For instance, runs conducted before the explicit

logging of a data path can be reliably attributed to the original, default data path used at that time.

5.6 Conclusion

We demonstrated that configuration validation remains a challenging task for both LLMs and RAG systems. We did not surpass the results of the Ciri team with any RAG configuration. In our experiments, retrieval appeared to be the primary bottleneck. Retrieving up-to-date documentation pages for system configurations was not effective using standard BM25, a general-purpose embedding retriever, or even the specialized one we tested. Retrieval techniques and parameters, such as hybrid retrieval, did not significantly impact retrieval quality. Future work could include metadata filtering, custom retrieval models, more advanced rewriting techniques, or adaptive RAG systems (also known as agentic systems), which were beyond the scope of our research.

Our findings indicate that more recent models are more capable of correctly validating configurations, yet we also observed that their performance varies, and the results are not yet as reliable as required for this task. We updated the Ciri experiment with more recent models; however, we could not establish a new state-of-the-art result. Ciri's highest F1-score (0.790) was reported with Claude-3-Sonnet, a model not accessible during our experiments. Our application of DeepSeek's *DeepSeek-chat-v3-0324* model using the Ciri configuration yielded a comparable F1-score of 0.776.

Chapter 6

Conclusion

The advent of Large Language Models (LLMs) has revolutionized natural language processing, yet their inherent limitations, such as knowledge cutoffs, potential for hallucination, and difficulty accessing private or rapidly changing information, necessitate more robust solutions for many real-world applications. Retrieval-Augmented Generation (RAG) systems have emerged as a promising approach to mitigate these issues by grounding LLM responses in external knowledge sources. However, as explored throughout this thesis, this approach presents its own significant trade-offs. The integration of retrieval mechanisms introduces significant complexities, increased inference times, and new challenges in evaluation and optimization.

This thesis addressed these challenges by considering two central questions: "Is an advanced RAG system necessary for a given use case, or is a standalone LLM sufficient?" and "How can one determine if a specific RAG configuration is optimal for its intended task?" The primary contribution of this work is the development and demonstration of RAGnRoll, a comprehensive benchmarking framework designed to facilitate systematic and reproducible RAG evaluation.

RAGnRoll, detailed in Chapter 4, provides a structured methodology to navigate the intricacies of RAG development. Key features of the framework include:

- **Systematic Baselining:** The framework mandates the evaluation of standalone LLMs and naive RAG configurations as baselines (as argued in Section 4.2.1). This crucial step helps quantify the added value of more complex RAG architectures and informs the decision of whether RAG is beneficial for the specific task and dataset.
- **Rigorous Validation and Generalization Assessment:** Inspired by standard machine learning practices, RAGnRoll enforces a strict validation-test split of evaluation data (Section 4.1). All system tuning and iterative

reconfiguration are performed exclusively on the validation set, with the test set reserved for a final assessment of generalization error, thereby mitigating the risk of overfitting to the development data, a critical concern when tuning complex RAG systems.

- **Comprehensive Multi-faceted Evaluation:** Recognizing that RAG performance is multi-dimensional, the framework supports both end-to-end evaluation and granular component-level analysis. This allows for the identification of bottlenecks within the RAG pipeline. Furthermore, this thesis argues for and the framework design accommodates the inclusion of hardware metrics (e.g., latency, resource utilization) alongside traditional accuracy-based metrics, providing a holistic view of a system's practicality.

- **Transparent and Reproducible Experimentation:** Leveraging tools like MLflow for results logging and Langfuse for detailed execution tracing (Section 4.6), RAGnRoll promotes transparency and aids in in-depth failure analysis. The use of Haystack's declarative pipeline configurations further enhances reproducibility.

The practical utility and effectiveness of the RAGnRoll framework were demonstrated in Chapter 5 through an experiment focused on software configuration validation. This application showcased how the framework can guide the iterative development and comparison of different RAG systems, leading to optimized configurations tailored to specific requirements, including considerations for output formatting and computational overhead.

The practical application in Chapter 5 served to demonstrate RAGnRoll's core functionalities. However, the breadth of RAG configurations explored in that instance was necessarily guided by the project's timeframe and resources. The reported experiments, with an approximate cost of 200 Euro, highlight the framework's utility for initial, cost-effective investigations. A key strength of RAGnRoll is its design for scalability; with more extensive time and financial backing, it can be readily employed to systematically evaluate a far wider spectrum of embedding models, generation models, and intricate RAG architectures than was feasible within the direct experimental scope of this thesis. This underscores its potential for larger-scale research or industrial deployment.

In conclusion, this thesis provides both a conceptual blueprint and a practical implementation for the robust evaluation of RAG systems. By emphasizing systematic comparison, rigorous testing for generalization, comprehensive metric collection, and detailed failure analysis, the RAGnRoll framework empowers researchers and practitioners to make data-driven decisions in the rapidly

evolving landscape of retrieval-augmented generation. It offers a path to move beyond ad-hoc tuning towards a more principled approach to building effective and efficient RAG solutions, ultimately helping to answer whether a RAG system is indeed necessary and how to best configure it for a specific use case.

Appendix A

Pipeline Diagrams

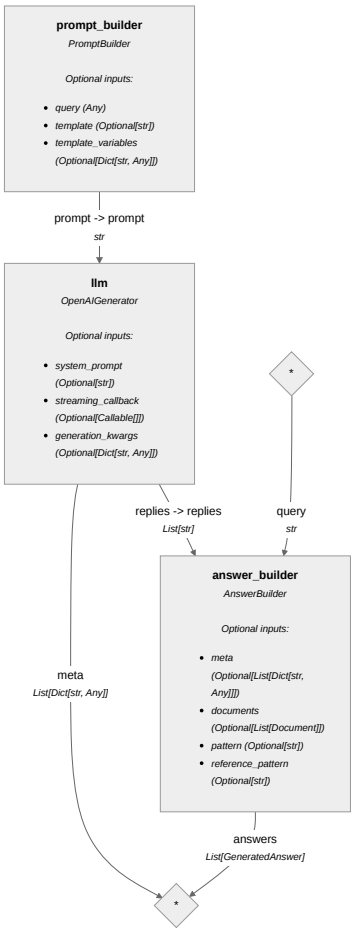

Figure A.1: Pipeline diagram of the standalone LLM baseline implementation

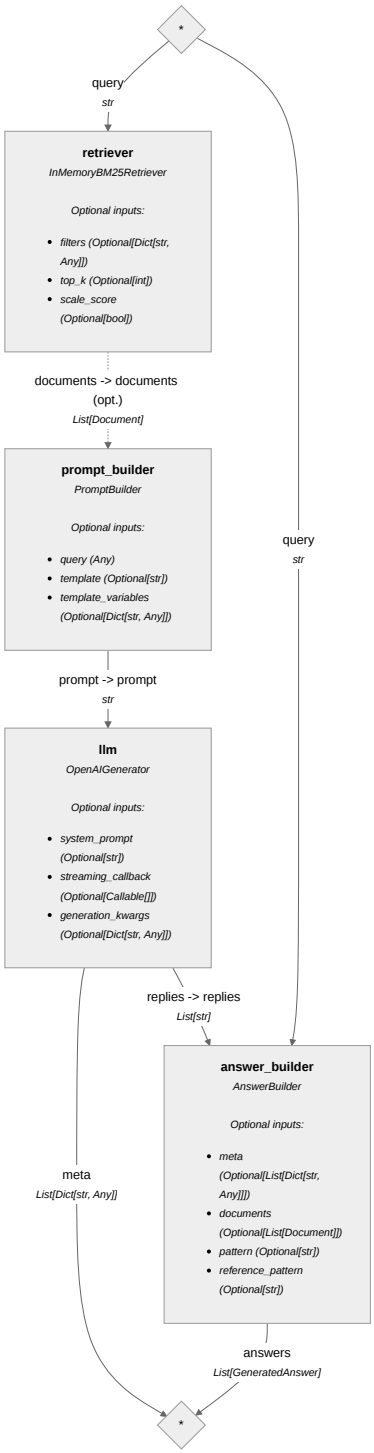

Figure A.2: Pipeline diagram of the BM25 baseline implementation

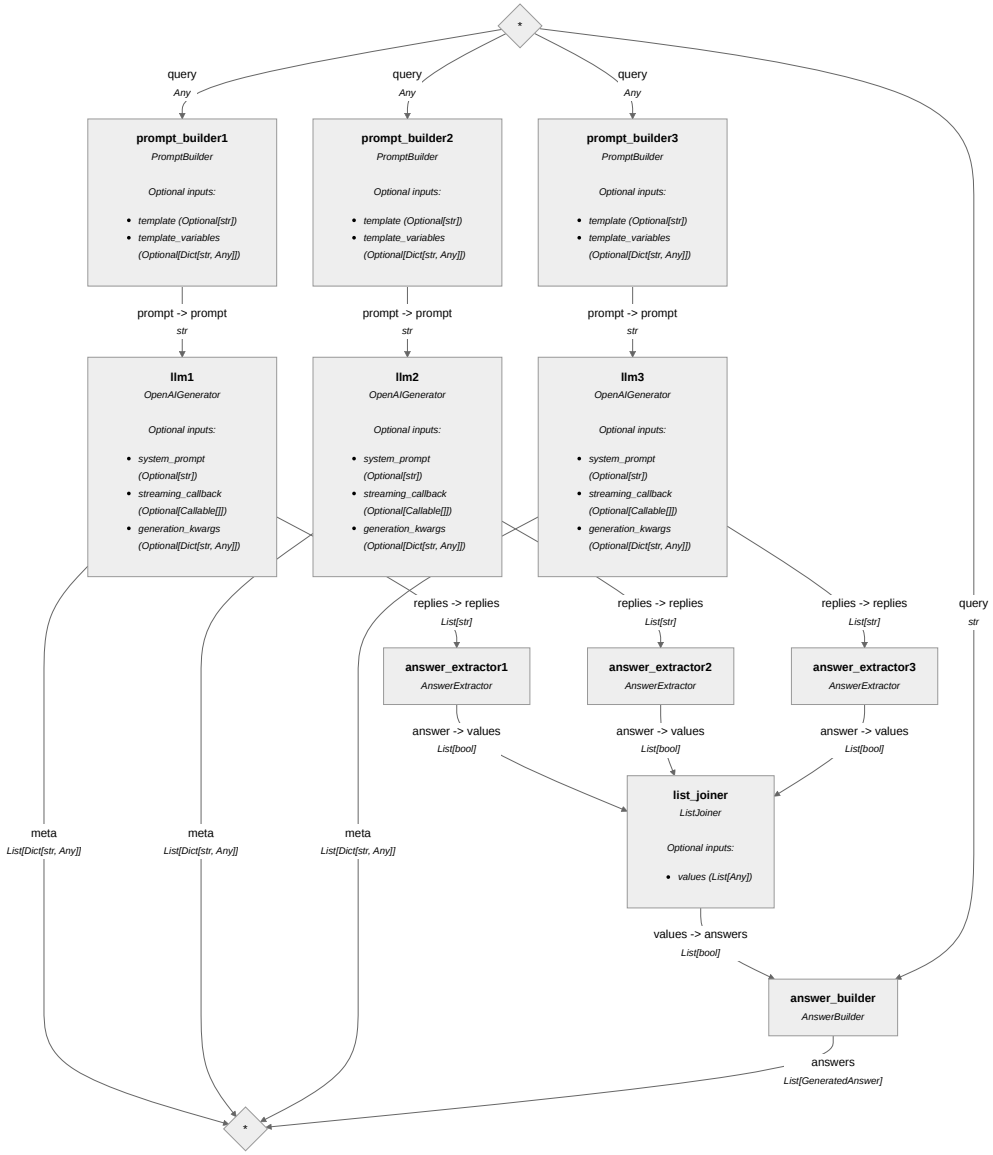

Figure A.3: Pipeline diagram of the Ciri baseline implementation

Bibliography

[1] D. Ackley, G. Hinton, and T. Sejnowski. A learning algorithm for boltzmann machines. *Cognitive Science*, 9(1):147–169, 1985. ISSN 03640213. doi: 10.1016/S0364-0213(85)80012-4. 2.2

[2] Jan Albrecht. GitHub - AlJ95/master-thesis-alj95, 2025. URL `https://github.com/AlJ95/master-thesis-alj95`. 4.6

[3] Anthropic. Introducing claude, Mar 2023. URL `https://www.anthropic.com/news/introducing-claude`. 1

[4] Ashkan Alinejad, Krtin Kumar, and Ali Vahdat. Evaluating the retrieval component in llm-based question answering systems. *arXiv.org*, 2024. URL `https://www.semanticscholar.org/paper/Evaluating-the-Retrieval-Component-in-LLM-Based-Alinejad-Kumar/d274dccfae844b06de6a2d7d083b264749adc599`. 3.1

[5] Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. Seven failure points when engineering a retrieval augmented generation system. In Jan Bosch, Grace Lewis, Jane Cleland-Huang, and Henry Muccini, editors, *Proceedings of the IEEE/ACM 3rd International Conference on AI Engineering - Software Engineering for AI*, pages 194–199, New York, NY, USA, 2024. ACM. ISBN 9798400705915. doi: 10.1145/3644815.3644945. 3.1, 4

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. URL `http://arxiv.org/pdf/2005.14165`. 2.1.3

[7] Yupeng Chang, Xu Wang, Jindong Wang, Yuan Wu, Linyi Yang, Kaijie Zhu, Hao Chen, Xiaoyuan Yi, Cunxiang Wang, Yidong Wang, Wei Ye, Yue Zhang, Yi Chang, Philip S. Yu, Qiang Yang, and Xing Xie. A survey on evaluation of large language models. URL `http://arxiv.org/pdf/2307.03109`. 3.1

[8] Harrison Chase. LangChain. `https://github.com/langchain-ai/langchain`, October 2022. URL `https://github.com/langchain-ai/langchain`. 1, 2.2.1, 2.3.2, 4.3

[9] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. Benchmarking large language models in retrieval-augmented generation. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(16):17754–17762, 2024. ISSN 2159-5399. doi: 10.1609/aaai.v38i16.29728. 2.3

[10] Cheng-Han Chiang and Hung-yi Lee. Can large language models be an alternative to human evaluations? pages 15607–15631, July 2023. doi: 10.18653/v1/2023.acl-long.870. URL `https://aclanthology.org/2023.acl-long.870/`. 4.2.2, 4.2.2

[11] Chroma, 2025. URL `https://trychroma.com/`. 4.3

[12] DeepL. Deepl übersetzer: Der präziseste übersetzer der welt, 2017. URL `https://www.deepl.com/de/`. 1

[13] DeepSeek-AI. Deepseek-v3 technical report, 2024. URL `https://arxiv.org/abs/2412.19437`. 5.3.2, 5.3.2

[14] deepset GmbH. Deepset studio: Sign up, 2025. URL `https://www.deepset.ai/deepset-studio`. 4.3

[15] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL `https://arxiv.org/abs/1810.04805`. 2.2.2

[16] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. Ragas: Automated evaluation of retrieval augmented generation, 2023. URL `https://arxiv.org/abs/2309.15217`. 3.2

[17] Evidently AI, Inc. Mean average precision (map) in ranking and recommendations, 2025. URL `https://www.evidentlyai.com/ranking-metrics/mean-average-precision-map`. 4.2.2

[18] Kshitij Fadnis, Siva Sankalp Patel, Odellia Boni, Yannis Katsis, Sara Rosenthal, Benjamin Sznajder, and Marina Danilevsky. Inspectorraget: An introspection platform for rag evaluation. URL http://arxiv.org/pdf/2404.17347. 3.2

[19] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation, 2018. URL http://arxiv.org/pdf/1805.04833. 2.1.2

[20] Nat Friedman. Introducing github copilot: Your ai pair programmer, Feb 2022. URL https://github.blog/news-insights/product-news/introducing-github-copilot-ai-pair-programmer/. 1

[21] 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, 2023. URL http://arxiv.org/pdf/2312.10997. 2.2.2, 2.3, 2.3.1, 2.3.1, 2.3.2, 2.3.2, 2.3.2, 2.4, 3.1, 4.3

[22] Inc. GitLab. The most-comprehensive AI-powered DevSecOps platform, 2025. URL https://about.gitlab.com/. 4.4

[23] Hetzner Online GmbH. Cheap dedicated servers, cloud and hosting from Germany, 2025. URL https://www.hetzner.com/. 5.3.1

[24] Ari Holtzman, Jan Buys, Du Li, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2019. URL http://arxiv.org/pdf/1904.09751. 2.1.2

[25] Yasuto Hoshi, Daisuke Miyashita, Youyang Ng, Kento Tatsuno, Yasuhiro Morioka, Osamu Torii, and Jun Deguchi. Ralle: A framework for developing and evaluating retrieval-augmented large language models. URL http://arxiv.org/pdf/2308.10633v2. 3.2

[26] Yasuto Hoshi, Daisuke Miyashita, Youyang Ng, Kento Tatsuno, Yasuhiro Morioka, Osamu Torii, and Jun Deguchi. Ralle: A framework for developing and evaluating retrieval-augmented large language models. https://arxiv.org/abs/2308.10633, 2023. URL https://arxiv.org/abs/2308.10633. 1

[27] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, Lo David, John Grundy, and Haoyu Wang. Large language models for software engineering: A systematic literature review, 2023. URL http://arxiv.org/pdf/2308.10620v6. 4.1

[28] Jennifer Hsia, Afreen Shaikh, Zhiruo Wang, and Graham Neubig. Ragged: Towards informed design of retrieval augmented generation systems. *arXiv.org*, 2024. URL https://www.semanticscholar.org/paper/RAGGED%3A-Towards-Informed-Design-of-Retrieval-Hsia-Shaikh/e1446fc9b11e73e9f1d867df9012fdc3d3df60d0. 3.2

[29] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *ACM Transactions on Information Systems*, 43(2):1–55, January 2025. ISSN 1558-2868. doi: 10.1145/3703155. URL http://dx.doi.org/10.1145/3703155. 2.3, 3.1, 4.2.2, 4.2.2

[30] Peng Huang, William J. Bolosky, Abhishek Singh, and Yuanyuan Zhou. Confvalley. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, New York, NY, USA, 2015. ACM. doi: 10.1145/2741948.2741963. 5.2

[31] Inc. Hugging Face. MTEB Leaderboard - a Hugging Face Space by mteb, 2024. URL https://huggingface.co/spaces/mteb/leaderboard. 5.3.2

[32] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. 5.3.2

[33] DataChain Inc. User guide | data version control · dvc, 2020. URL https://dvc.org/doc/user-guide. 4.4

[34] Finto Technologies Inc. Open source llm engineering platform, 2022. URL https://langfuse.com/. 4.2, 4.2.2

[35] Github Inc. GitHub · Build and ship software on a single, collaborative platform, 2025. URL https://github.com/. 4.4

[36] Peter Izsak, Moshe Berchansky, Daniel Fleischer, and Ronen Laperdon. fastRAG: Efficient Retrieval Augmentation and Generation Framework. https://github.com/IntelLabs/fastrag, 2023. URL https://github.com/IntelLabs/fastrag. 1

[37] Bowen Jin, Jinsung Yoon, Jiawei Han, and Sercan O. Arik. Long-context llms meet rag: Overcoming challenges for long inputs in rag, 2024. URL https://arxiv.org/abs/2410.05983. 3.1

[38] Jiajie Jin, Yutao Zhu, Xinyu Yang, Chenghao Zhang, and Zhicheng Dou. Flashrag: A modular toolkit for efficient retrieval-augmented generation research. URL `http://arxiv.org/pdf/2405.13576v1`. 3.2, 4.2.2

[39] Jiajie Jin, Yutao Zhu, Xinyu Yang, Chenghao Zhang, and Zhicheng Dou. Flashrag: A modular toolkit for efficient retrieval-augmented generation research. *CoRR*, abs/2405.13576, 2024. URL `https://arxiv.org/abs/2405.13576`. 1

[40] Nikhil Kandpal, Haikang Deng, Adam Roberts, Eric Wallace, and Colin Raffel. Large language models struggle to learn long-tail knowledge, 2022. URL `http://arxiv.org/pdf/2211.08411`. 1

[41] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering, 2020. URL `https://arxiv.org/abs/2004.04906`. 2.2.2

[42] Dongkyu Kim, Byoungwook Kim, Donggeon Han, and Matouš Eibich. Autorag: Automated framework for optimization of retrieval augmented generation pipeline. URL `http://arxiv.org/pdf/2410.20878v1`. 3.2

[43] Jeffrey Kim and Bobb Kim. AutoRAG. `https://github.com/Marker-Inc-Korea/AutoRAG`, February 2024. URL `https://github.com/Marker-Inc-Korea/AutoRAG`. 1

[44] Satyapriya Krishna, Kalpesh Krishna, Anhad Mohananey, Steven Schwarcz, Adam Stambler, Shyam Upadhyay, and Manaal Faruqui. Fact, fetch, and reason: A unified evaluation of retrieval-augmented generation. URL `http://arxiv.org/pdf/2409.12941`. 3.1

[45] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. URL `http://arxiv.org/pdf/2309.06180`. 5.3.1

[46] LLC LF Projects. Mlflow, 2018. URL `https://mlflow.org/`. 4.2

[47] Dawei Li, Renliang Sun, Yue Huang, Ming Zhong, Bohan Jiang, Jiawei Han, Xiangliang Zhang, Wei Wang, and Huan Liu. Preference leakage: A contamination problem in llm-as-a-judge, . URL `http://arxiv.org/pdf/2502.01534`. 4.5

[48] Siran Li, Linus Stenzel, Carsten Eickhoff, and Seyed Ali Bahrainian. Enhancing retrieval-augmented generation: A study of best practices, . URL http://arxiv.org/pdf/2501.07391. 3.1, 3.1, 3.1

[49] Xinyu Lian, Yinfang Chen, Runxiang Cheng, Jie Huang, Parth Thakkar, Minjia Zhang, and Tianyin Xu. Configuration validation with large language models, 2024. URL https://arxiv.org/abs/2310.09690. 5.1, 5.2, 5.3, 5.3.1

[50] Jimmy Lin, Rodrigo Nogueira, and Andrew Yates. Pretrained transformers for text ranking: Bert and beyond. URL http://arxiv.org/pdf/2010.06467. 4.2.2

[51] Jerry Liu. LlamaIndex. https://github.com/jerryjliu/llama_index, 11 2022. URL https://github.com/jerryjliu/llama_index. 1, 2.2.1, 2.3.2, 4.3

[52] Jintao Liu, Ruixue Ding, Linhao Zhang, Pengjun Xie, and Fie Huang. Cofe-rag: A comprehensive full-chain evaluation framework for retrieval-augmented generation with enhanced data diversity. *arXiv.org*, 2024. URL https://www.semanticscholar.org/paper/CoFE-RAG%3A-A-Comprehensive-Full-chain-Evaluation-for-Liu-Ding/779797f595acb8dc35690c1252a8b415455a3ded. 3.1

[53] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. URL http://arxiv.org/pdf/2307.03172. 3.1, 4.2.3, 5.3.2

[54] Alex Mallen, Akari Asai, Victor Zhong, Rajarshi Das, Daniel Khashabi, and Hannaneh Hajishirzi. When not to trust language models: Investigating effectiveness of parametric and non-parametric memories. URL http://arxiv.org/pdf/2212.10511. 2.4, 3.1, 4.2.2

[55] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to information retrieval*. Cambridge Univ. Press, Cambridge, reprinted. edition, 2009. ISBN 9780521865715. doi: 10.1017/CBO9780511809071. 2.2.1, 2.2.1

[56] Niklas Muennighoff, Nouamane Tazi, Loïc Magne, and Nils Reimers. Mteb: Massive text embedding benchmark. *arXiv preprint arXiv:2210.07316*, 2022. doi: 10.48550/ARXIV.2210.07316. URL https://arxiv.org/abs/2210.07316. 2.2.2, 5.3.2

[57] OpenAI, Nov 2022. URL `https://openai.com/index/chatgpt`. 1, 4.5, 5.3.2, 5.3.2, 5.3.2, 5.3.2

[58] Inc OpenRouter. OpenRouter, 2023. URL `https://openrouter.ai/`. 5.3.1

[59] Sundar Pichai, Kathy Korevec, and Shrestha Basu Mallick. Introducing Gemini 2.0: our new AI model for the agentic era. 12 2024. URL `https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/`. 5.3.2

[60] Malte Pietsch, Timo Möller, Bogdan Kostic, Julian Risch, Massimiliano Pippi, Mayank Jobanputra, Sara Zanzottera, Silvano Cerza, Vladimir Blagojevic, Thomas Stadelmann, Tanay Soni, and Sebastian Lee. Haystack: the end-to-end NLP framework for pragmatic builders. `https://github.com/deepset-ai/haystack`, November 2019. URL `https://github.com/deepset-ai/haystack`. 1, 4.2.2, 4.3

[61] Marco AF Pimentel, Clément Christophe, Tathagata Raha, Prateek Munjal, Praveen K Kanithi, and Shadab Khan. Beyond metrics: A critical analysis of the variability in large language model evaluation frameworks. 2024. URL `https://arxiv.org/abs/2407.21072`. 3.1

[62] Qdrant. QDRant - Vector Database, 2025. URL `https://qdrant.tech/`. 4.3

[63] Hannah Rashkin, David Reitter, Gaurav Singh Tomar, and Dipanjan Das. Increasing faithfulness in knowledge-grounded dialogue with controllable features. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 704–718, Online, August 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.acl-long.58. URL `https://aclanthology.org/2021.acl-long.58/`. 2.3

[64] David Rau, Hervé Déjean, Nadezhda Chirkova, Thibault Formal, Shuai Wang, Vassilina Nikoulina, and Stéphane Clinchant. Bergen: A benchmarking library for retrieval-augmented generation. URL `http://arxiv.org/pdf/2407.01102`. 3.2

[65] Dongyu Ru, Lin Qiu, Xiangkun Hu, Tianhang Zhang, Peng Shi, Shuaichen Chang, Cheng Jiayang, Cunxiang Wang, Shichao Sun, Huanyu Li, Zizhao Zhang, Binjie Wang, Jiarong Jiang, Tong He, Zhiguo Wang, Pengfei Liu,

Yue Zhang, and Zheng Zhang. Ragchecker: A fine-grained framework for diagnosing retrieval-augmented generation. URL `http://arxiv.org/pdf/2408.08067`. 3.1, 3.2

[66] RunPod. RunPod - the cloud built for AI, 2025. URL `https://www.runpod.io/`. 5.3.1

[67] Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. Ares: An automated evaluation framework for retrieval-augmented generation systems. URL `http://arxiv.org/pdf/2311.09476`. 3.2

[68] Alireza Salemi and Hamed Zamani. Evaluating retrieval quality in retrieval-augmented generation. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '24, page 2395–2400, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400704314. doi: 10.1145/3626772.3657957. URL `https://doi.org/10.1145/3626772.3657957`. 1, 3.1, 3.1, 4.2.2

[69] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, Pranav Sandeep Dulepet, Saurav Vidyadhara, Dayeon Ki, Sweta Agrawal, Chau Pham, Gerson Kroiz, Feileen Li, Hudson Tao, Ashay Srivastava, Hevander Da Costa, Saloni Gupta, Megan L. Rogers, Inna Goncearenco, Giuseppe Sarli, Igor Galynker, Denis Peskoff, Marine Carpuat, Jules White, Shyamal Anadkat, Alexander Hoyle, and Philip Resnik. The prompt report: A systematic survey of prompting techniques. URL `http://arxiv.org/pdf/2406.06608`. 2.1.3, 2.1.3

[70] Kurt Shuster, Spencer Poff, Moya Chen, Douwe Kiela, and Jason Weston. Retrieval augmentation reduces hallucination in conversation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 3784–3803, Stroudsburg, PA, USA, 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.320. 2.3

[71] Sebastian Simon, Alina Mailach, Johannes Dorn, and Norbert Siegmund. A methodology for evaluating rag systems: A case study on configuration dependency validation, 2024. URL `http://arxiv.org/pdf/2410.08801v1`. 1, 3.1, 4, 4.4

[72] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context

to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/sun. 5.2

[73] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In Ethan L. Miller, editor, *Proceedings of the 25th Symposium on Operating Systems Principles*, ACM Digital Library, pages 328–343, New York, NY, 2015. ACM. ISBN 9781450338349. doi: 10.1145/2815400.2815401. 5.1

[74] Qwen Team. Qwen3, April 2025. URL https://qwenlm.github.io/blog/qwen3/. 5.3.2

[75] Qwen Team. Qwq-32b: Embracing the power of reinforcement learning, March 2025. URL https://qwenlm.github.io/blog/qwq-32b/. 5.3.2

[76] Shangqing Tu, Yuanchun Wang, Jifan Yu, Yuyang Xie, Yaran Shi, Xiaozhi Wang, Jing Zhang, Lei Hou, and Juanzi Li. R-eval: A unified toolkit for evaluating domain knowledge of retrieval augmented large language models. URL http://arxiv.org/pdf/2406.11681. 3.2

[77] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017, revised in 2023. URL https://arxiv.org/abs/1706.03762. 1, 2.1

[78] Stefan Wagner, Marvin Muñoz Barón, Davide Falessi, and Sebastian Baltes. Towards evaluation guidelines for empirical studies involving llms. URL http://arxiv.org/pdf/2411.07668. 3.1

[79] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Le Quoc, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. URL http://arxiv.org/pdf/2201.11903. 2.1.3

[80] Wikipedia. Leipzig, 2025. URL https://en.wikipedia.org/w/index.php?title=Leipzig&oldid=1269825057. 2.3.2, 2.7

[81] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger,

Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Huggingface's transformers: State-of-the-art natural language processing, 2019. URL `http://arxiv.org/pdf/1910.03771`. 2.1

[82] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations. In *Proceedings of the 2020 USENIX Annual Technical Conference*. USENIX Association, 2020, [Berkeley, CA], 2020. ISBN 9781939133144. 5.1

[83] xlab uiuc and Lian [xylian86] Xinyu. GitHub - xlab-uiuc/ciri, 1 2025. URL `https://github.com/xlab-uiuc/ciri/tree/main`. 5.3.1

[84] Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, Chenyang Shao, Yuwei Yan, Qinglong Yang, Yiwen Song, Sijian Ren, Xinyuan Hu, Yu Li, Jie Feng, Chen Gao, and Yong Li. Towards large reasoning models: A survey of reinforced reasoning with large language models. URL `http://arxiv.org/pdf/2501.09686`. 4.2.2

[85] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zhihao Fan. Qwen2 technical report. *arXiv preprint arXiv:2407.10671*, 2024. 5.3.2

[86] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tianyi Tang, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*, 2024. 5.3.2

[87] Junhan Yang, Jiahe Wan, Yichen Yao, Wei Chu, Yinghui Xu, and Yuan Qi. inf-retriever-v1 (revision 5f469d7), 2025. URL `https://huggingface.co/infly/inf-retriever-v1`. 5.3.2

[88] Junqi Yin, Avishek Bose, Guojing Cong, Isaac Lyngaas, and Quentin Anthony. Comparative study of large language model architectures on frontier. In Kyle Chard and Zhuozhao Li, editors, *2024 IEEE International Parallel and Distributed Processing Symposium*, pages 556–569, Piscataway, NJ, 2024. IEEE. ISBN 979-8-3503-8711-7. doi: 10.1109/IPDPS57955.2024.00056. 2.1.1

[89] Hao Yu, Aoran Gan, Kai Zhang, Shiwei Tong, Qi Liu, and Zhaofeng Liu. Evaluation of retrieval-augmented generation: A survey, 2024. URL `https://arxiv.org/abs/2405.07437`. 1, 2.3, 3.1, 3.1, 4.2.2

[90] Zhengran Zeng, Yidong Wang, Rui Xie, Wei Ye, and Shikun Zhang. Coderujb: An executable and unified java benchmark for practical programming scenarios. URL `http://arxiv.org/pdf/2403.19287`. 4.1

[91] Xuanwang Zhang, Yunze Song, Yidong Wang, Shuyun Tang, Xinfeng Li, Zhengran Zeng, Zhen Wu, Wei Ye, Wenyuan Xu, Yue Zhang, Xinyu Dai, Shikun Zhang, and Qingsong Wen. Raglab: A modular and research-oriented unified framework for retrieval-augmented generation. URL `http://arxiv.org/pdf/2408.11381v2`. 3.2

[92] Xuanwang Zhang, Yun-Ze Song, Yidong Wang, Shuyun Tang, Xinfeng Li, Zhengran Zeng, Zhen Wu, Wei Ye, Wenyuan Xu, Yue Zhang, Xinyu Dai, Shikun Zhang, and Qingsong Wen. RAGLAB: A modular and research-oriented unified framework for retrieval-augmented generation. In Delia Irazu Hernandez Farias, Tom Hope, and Manling Li, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 408–418, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-demo.43. URL `https://aclanthology.org/2024.emnlp-demo.43/`. 1

[93] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. Siren's song in the ai ocean: A survey on hallucination in large language models, 2023. URL `http://arxiv.org/pdf/2309.01219`. 1

[94] 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, 2024. URL `http://arxiv.org/pdf/2402.19473`. 2.2, 3.1, 3.1