



UNIVERSITÀ
DI TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Artificial Intelligence Systems

FINAL DISSERTATION

EVALUATING
LARGE LANGUAGE MODELS FOR
RETRIEVAL-AUGMENTED GENERATION
IN ENTERPRISE APPLICATIONS

*An Evaluation Pipeline for
Nokia's "Network as Code" SDK Documentation Assistant*

Supervisor
Staiano Jacopo

Student
Canova Tommaso

Academic year 2023/2024

Contents

Abstract	3
1 Introduction	4
1.1 Motivation	4
1.2 Thesis outline	4
2 Background and Related Work	5
2.1 Foundations of Language Models	5
2.1.1 Foundational block: Transformers	5
2.1.2 Encoder-Only model: BERT	9
2.1.3 Decoder-only Model: GPT-1 and GPT-2	11
2.2 Large Language Models (LLMs)	12
2.2.1 Open source	13
2.2.2 Scaling laws	13
2.3 Retrieval Augmented Generation (RAG)	15
2.3.1 Vector Database Management Systems (VDBMS)	15
2.3.2 Rerankers	16
2.3.3 RAG paradigms	17
2.3.4 RAG vs Fine-Tuning	17
2.3.5 Prompt Engineering	18
2.4 LLM Evaluation	20
2.4.1 Traditional Natural Language Processing (NLP) evaluation	20
2.4.2 Types of Evaluation	23
2.4.3 LLM Benchmarks	25
2.4.4 LLM as Judge paradigm	27
2.4.5 RAG Evaluation	29
2.4.6 Challenges	31
2.5 Telecommunication concepts	33
2.5.1 5G Networks	33
2.5.2 Programmable Networks	33
2.5.3 Quality of Service on Demand	33
2.5.4 Network Slicing	33
3 Methodology	35
3.1 Setup	35
3.1.1 Data source: Network as Code (NaC) SDK documentation	35
3.1.2 Data gathering with Copilot (Nokia's internal assistant)	36
3.1.3 Additional Tools Utilized	41
3.2 Evaluation Tool	42
3.2.1 NaC Sandbox Evaluator	43
3.2.2 NaC Evaluation Charts	49

4 Results	55
4.1 Overview of Aggregate Performance Results	55
4.2 Performance Evaluation of Individual Models	57
4.3 Comparative Analysis of Metrics	63
4.4 Common Patterns Observed in LLM Outputs	64
5 Limitations and Future Works	65
6 Conclusion and discussion	67
Bibliography	69
List of Acronyms	81

Abstract

This thesis presents a comprehensive evaluation framework for Large Language Models (LLMs), specifically tailored for business applications within a Retrieval-Augmented Generation (RAG) pipeline. The core of this evaluation tool is a "three pillars" framework, combining *(i)* Text Similarity, *(ii)* Code Evaluation, and *(iii)* LLM-as-Judge assessments. This design enables a multifaceted analysis of LLM performance by examining relevance, code quality, and judgment accuracy, adaptable to various knowledge bases beyond the tested Network as Code SDK documentation.

In this work, 18 LLMs were benchmarked against *LLaMA 3.1 70B Instruct*, revealing significant performance variations in code generation, with *GPT-4o* leading in accuracy and others demonstrating challenges, particularly in asynchronous code generation. The results questioned the reliability of traditional metrics like BERT Score in detecting quality nuances and highlighted the predictive power of CodeBLEU for code-specific outcomes. Additionally, the study underscored the potential of LLM-as-Judge for evaluating relevance and hallucination, though human supervision remains essential.

The thesis identifies challenges in applying LLMs to complex, real-world tasks, especially in specialized coding contexts, and outlines recommendations for future iterations. These include optimizing RAG parameters, testing extended context windows, and enhancing prompt engineering, particularly with closed-source models constrained by resource limits. This tool, now implemented within Nokia, offers an objective, time-efficient method for LLM assessment, establishing a robust basis for ongoing research and development in LLM evaluation.

1 Introduction

1.1 Motivation

The motivation behind this thesis stems from the development of a tool designed to evaluate the effectiveness of various Large Language Models (LLMs) in assisting both technical and non-technical users with the *Nokia "Network as Code" Software Development Kit (SDK)*.

At the time of writing, in the fall of 2024, we are witnessing an unprecedented surge in interest around artificial intelligence, often referred to as the "AI hype." This trend is largely driven by the accessibility and perceived usability of LLMs, which have made sophisticated AI capabilities available to a broad audience. For example, models such as *ChatGPT* [130] (released in 2022) have demonstrated that advanced AI can be harnessed by nearly anyone with internet access, although it should be noted that deploying and maintaining these models requires substantial computational resources, making them not entirely democratized.

While LLMs are indeed powerful tools, there remains a significant gap in the availability of universal methods for evaluating their performance. Academic research has provided valuable frameworks for assessing specific LLM capabilities in areas such as mathematical reasoning, translation accuracy, logical inference, programming proficiency, and conversational management. However, in more specialized or industry-specific applications—particularly those that fall outside the standard academic benchmarks—evaluating LLM performance becomes increasingly challenging.

This challenge underscores the need for a robust evaluation tool that offers clear, interpretable comparisons of LLM performance across different use cases. Such a tool would allow for an agnostic evaluation of any LLM based on the same task or scenario. A critical prerequisite for developing this tool is the creation of a "*golden dataset*" or *benchmark*, which serves as the foundation for crafting automated processes to calculate metrics and perform custom evaluations on the models. This thesis aims to address this need by developing an evaluation pipeline tailored for use with Nokia's Network as Code SDK (NaC SDK) documentation assistant.

1.2 Thesis outline

In *Background and Related Work* chapter, we introduce the foundational concepts and related studies essential to understanding the evaluation of Large Language Models within the context of Network as Code (NaC). This section provides the theoretical frameworks and relevant technologies that position this work within the broader landscape of LLM development and evaluation.

Methodology chapter describes the approach taken in this work, detailing the experimental setup, data sources, and tools specifically developed for evaluating LLM performance in NaC applications. The methods presented are designed to align with the unique technical requirements of NaC tasks.

The outcomes of the evaluation are presented in *Results*. Here, we analyze model performance across multiple metrics, examining the strengths and limitations observed in the context of NaC. Comparative analyses help to contextualize model behaviors and reveal patterns across different evaluation criteria.

In *Limitations and Future Works*, we discuss the limitations of this work and propose potential directions for future research. This section outlines areas for further development, including enhancements in data handling, evaluation techniques, and resource allocation to improve NaC-specific LLM evaluation.

Finally, *Conclusion and discussion* provides a summary of the main findings and contributions of this work, reflecting on its implications for LLM evaluation in specialized domains like NaC.

2 Background and Related Work

This section provides the necessary background to understand the thesis, offering a concise and accessible overview of some of the main foundational topics relevant to the work.

2.1 Foundations of Language Models

Language modeling has been a long-established area of research, tracing its roots back to the 1950s when Shannon first applied information theory to human language. Shannon's pioneering work introduced the concept of using statistical models to predict and compress natural language text, marking the inception of the first wave of language modeling with n-gram models [153]. Over the decades, language modeling has evolved through four distinct waves[122], each contributing to the foundation of modern natural language processing (*NLP*) systems.

First Wave: Statistical Language Models (SLMs)

The first wave, represented by statistical language models (*SLMs*) [76], focused on modeling text as a sequence of words, estimating the probability of each word based on its preceding words. The most common form of these models, n-gram models, utilized Markov chain processes to compute word probabilities. Despite their widespread use in various NLP tasks, n-gram models struggled with data sparsity, often requiring smoothing techniques to handle unseen words or sequences [21].

Second Wave: Neural Language Models (NLMs)

The second wave emerged with the development of neural language models (*NLMs*), which addressed the limitations of SLMs by mapping words into low-dimensional continuous vector spaces, known as word embeddings. NLMs used neural networks to predict the next word in a sequence by aggregating the embeddings of preceding words. This innovation not only alleviated data sparsity issues but also allowed for the computation of semantic similarity between different linguistic inputs [8]. However, early NLMs were primarily task-specific, limiting their general applicability.

Third Wave: Pre-trained Language Models (PLMs)

Pre-trained language models (*PLMs*) marked the third wave, introducing a paradigm shift through the pre-training and fine-tuning process. PLMs, such as *BERT* [37] and *GPT* [139], were pre-trained on large-scale text corpora for general tasks like word prediction, and later fine-tuned on specific tasks with smaller datasets. This task-agnostic approach enhanced the versatility and robustness of language models, allowing them to adapt to a wide range of NLP applications.

Fourth Wave: Large Language Models (LLMs)

The current and most transformative wave is defined by Large Language Models (*LLMs*), exemplified by models like *GPT-4* [131], *LLaMA 3* [42], *Mixtral* [70], *Gemma* [168], *Qwen2* [205] and *PaLM* [30]. These *transformer-based* models, containing billions of parameters, are trained on vast text corpora and exhibit advanced language understanding and generation capabilities. Notably, LLMs demonstrate emergent abilities such as in-context learning (being capable of any downstream tasks without any gradient update or fine-tuning), instruction following, and multi-step reasoning, which were absent in smaller models like PLMs. These abilities, along with augmentation through external tools and continual learning mechanisms, position LLMs as foundational components in the development of artificial general intelligence (AGI).

2.1.1 Foundational block: Transformers

Transformers are a neural network architecture that has revolutionized artificial intelligence since their introduction in the 2017 paper *Attention is All You Need* [176]. They are widely used in tasks such as text generation (e.g., *OpenAI's GPT*, *Meta's LLaMA*), image recognition, protein structure prediction, and game playing. Transformers are known for their versatility and power, especially due to their self-attention mechanism, which enables them to capture long-range dependencies in sequences.

Transformer consists of several key components: *Embedding*, *Transformer Block*, *Output Probabilities*. [29]

- **Embedding:** The input text is tokenized and transformed into numerical representations called embeddings. This process involves:
 1. **Tokenization:** The input text is broken down into smaller units called tokens, which can be words or subwords. Each token is assigned a unique ID from a predefined vocabulary.
 2. **Token Embedding:** Each token is mapped to a high-dimensional vector that captures its semantic meaning.
 3. **Positional Encoding:** To capture the order of tokens in the sequence, positional information is added to the token embeddings. This ensures that the model can differentiate between the same tokens appearing in different positions.
 4. **Final Embedding:** The token and positional embeddings are summed to produce the final embedding, which contains both the semantic meaning of the tokens and their position in the sequence.

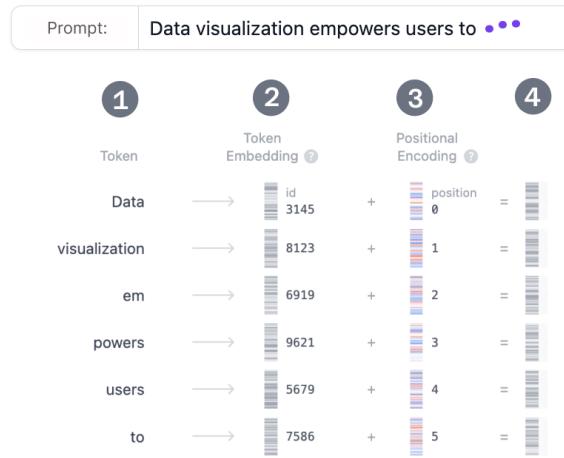


Figure 2.1: Embedding layer view that shows how the input prompt is converted to a vector representation. The process involves (1) Tokenization, (2) Token Embedding, (3) Positional Encoding, and (4) Final Embedding (Source: [Transformer Explained](#))

- **Transformer Block:** The core of the model consists of multi-head self-attention and a feed-forward *Multilayer Perceptron* (MLP) layer.
 1. **Multi-Head Self-Attention:** The self-attention mechanism enables the model to focus on relevant parts of the input sequence, capturing complex relationships between tokens. It consists of two steps.
 - **Step 1 - Query, Key, and Value Matrices computation:** Each token's embedding is transformed into three vectors: *Query (Q)*, *Key (K)*, and *Value (V)*. These vectors are derived by multiplying the input embedding matrix with learned weight matrices (Figure 2.2). An analogy to web search can help illustrate this:
 - * **Query (Q):** The search term you type into a search engine.
 - * **Key (K):** The titles of each webpage that the query can attend to.
 - * **Value (V):** The actual content of the most relevant webpages.
- Using these matrices, the model calculates attention scores, determining how much focus each token should receive.

Query, Key, Value Computation

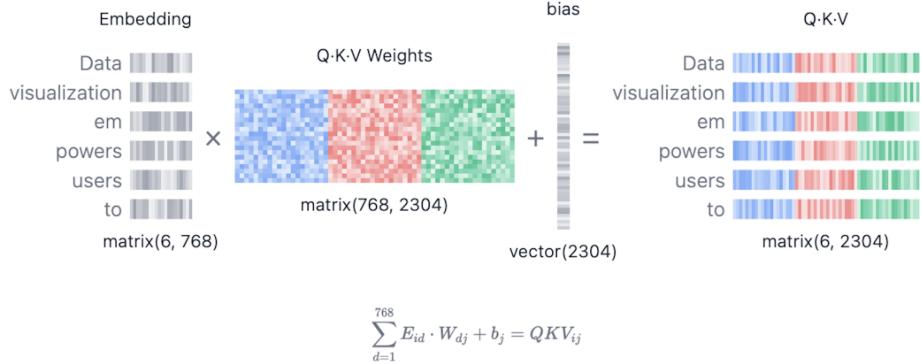


Figure 2.2: Visual computation of Query, Key and Value matrices from the original embedding (Source: [Transformer Explained](#))

- **Step 2: Masked Self-Attention:** During sequence generation, the model applies masked self-attention, preventing access to future tokens. The attention scores are calculated as the dot product of the Query and Key matrices, creating a relationship matrix between tokens. A mask is applied to prevent future token access by setting values to negative infinity. The *softmax* function then converts the attention scores into probabilities, highlighting the relevance of each token to the left of it. (Figure 2.3)

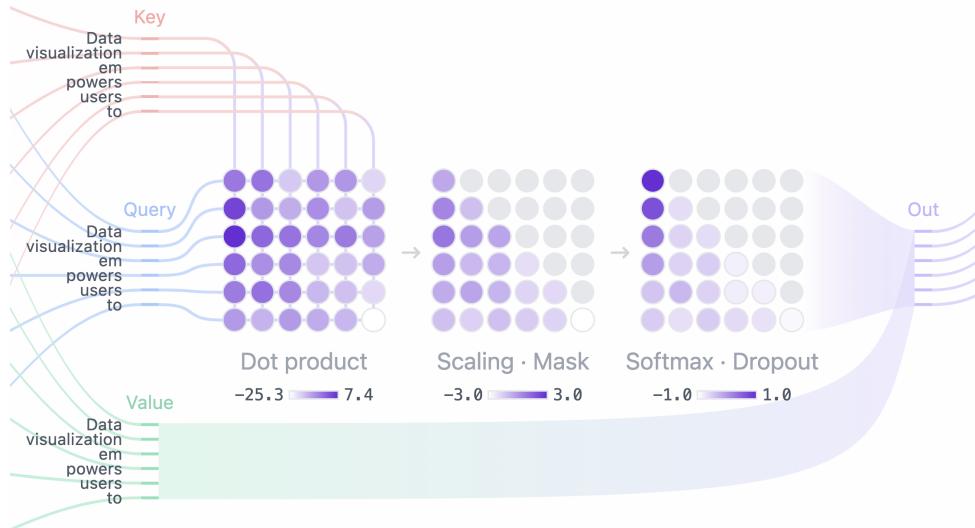


Figure 2.3: Masked self-attention computation (Source: [Transformer Explained](#))

2. **Multi-Layer Perceptron (MLP):** After the multiple heads of self-attention capture diverse relationships between the input tokens, the concatenated outputs are passed through the MLP layer to enhance the model's representational capacity. The MLP block consists of two linear transformations with a GELU activation function in between. The first linear transformation increases the dimensionality of the input, while the second transformation reduces it back to the original size, ensuring consistency for the subsequent layers. Unlike the self-attention mechanism, the MLP processes tokens independently, mapping them from one representation to another.

- **Output and Token Prediction:** After all the Transformer blocks, the output is passed through

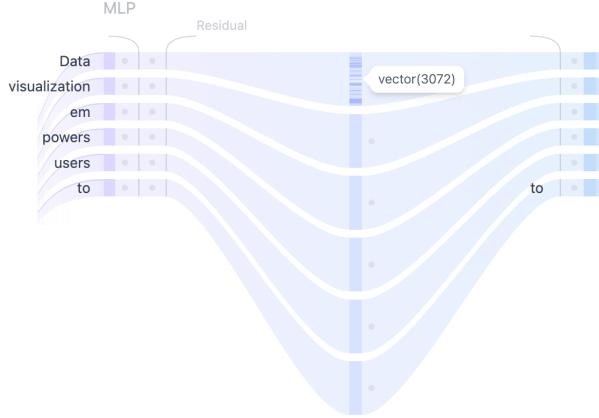


Figure 2.4: MLP layer that projects the self-attention representations into higher dimensions to enhance the model’s representational capacity (Source: [Transformer Explained](#)).

a final linear layer that projects the results in form of a 50’257 dimensional space, where each token has a corresponding value called a logit. The logits are ranked based on the likelihood of being the next token. The softmax function (Equation 2.5) is then applied to convert the logits into a probability distribution, from which the next token is sampled based on its likelihood. Finally, a hyperparameter called *temperature* is used in order to sample the next token from the distribution (Figure 2.6). From a mathematical perspective, the output logits are adjusted by dividing them by the temperature. The following outlines the effects of varying temperature values:

- **temperature = 1:** No effect on the softmax output; predictions are balanced.
- **temperature < 1:** Lowers the temperature, making the model more deterministic by sharpening the probability distribution.
- **temperature > 1:** Increases the temperature, softening the probability distribution and allowing for more random or creative outputs.

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^N e^{z_j}}$$

Figure 2.5: The Softmax function. It converts a vector of real-valued scores z_i into probabilities by normalizing the exponentials of the scores. Each output value lies between 0 and 1, and the sum of all outputs equals 1. This makes softmax useful for multi-class classification tasks in neural networks.

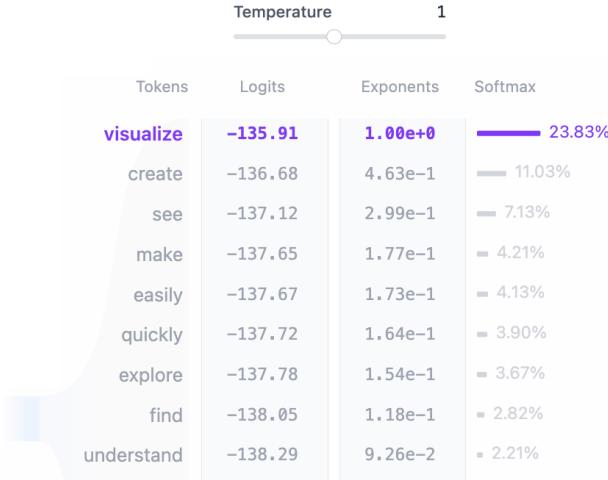


Figure 2.6: Probability of each token assigned considering output logits and temperature (Source: [Transformer Explained](#)).

Additionally, it is important to highlight that the following advanced architectural features have been key to unlocking the full potential of Transformers.

- **Layer Normalization:** Stabilizes training by normalizing inputs, ensuring consistent activations.
- **Dropout:** A regularization technique to prevent overfitting by randomly dropping units during training.
- **Residual Connections:** Allows gradients to flow through the network easily, preventing the vanishing gradient problem.

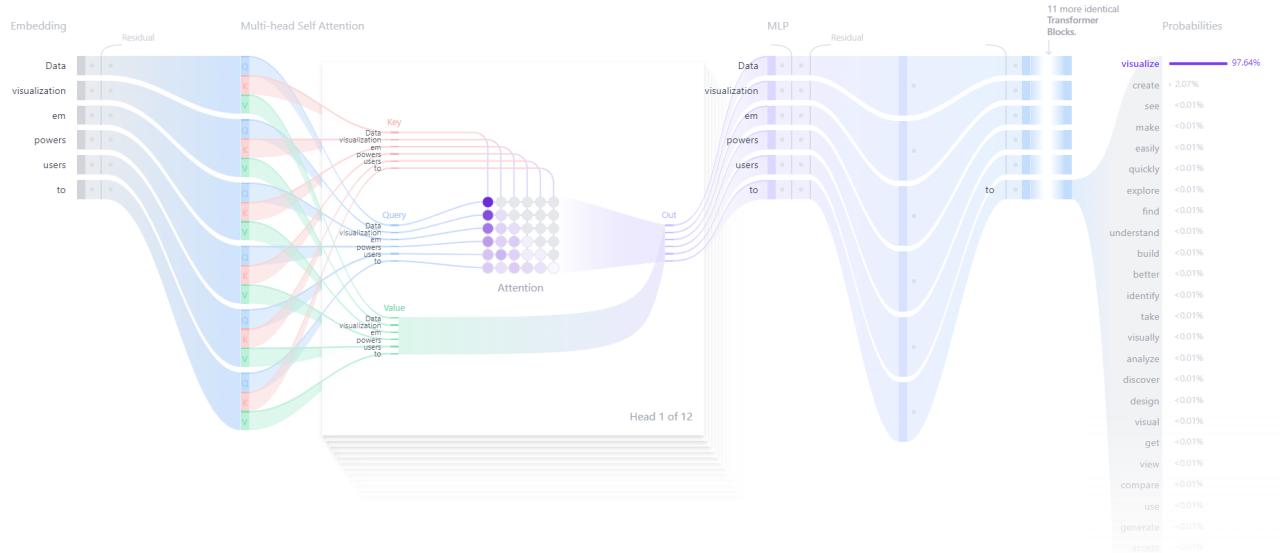


Figure 2.7: Transformers architecture visualized (Source: [Transformer Explained](#))

2.1.2 Encoder-Only model: BERT

In 2018, Google proposed a paradigm shift in Natural Language Processing with the introduction of the **BERT** (Bidirectional Encoder Representations from Transformers) model [37]. This groundbreaking architecture, based on the Transformer model, achieved state-of-the-art performance by leveraging

extensive pre-training on large-scale text corpora, encompassing approximately 3.3 billion words. Following this, it underwent fine-tuning for various downstream tasks, including question answering and text pair completion. Two primary versions of the model emerged: **BERT_{BASE}** and **BERT_{LARGE}**, as detailed in Table 2.1.

Distinct from OpenAI’s GPT-1 [139], which processes text sequentially in a left-to-right manner, BERT utilizes the full context of both preceding and subsequent words, thereby enhancing comprehension through its inherent bidirectionality.

Model	L (Layers)	H (Hidden Size)	A (Attention Heads)	Total Parameters
BERT_{BASE}	12	768	12	110M
BERT_{LARGE}	24	1024	16	340M

Table 2.1: BERT model available architectures from [37]

During the initial phase of pre-training, BERT engages in intensive training through *Masked Language Modeling (MLM)*, wherein 15% of the tokens in each sequence are randomly masked to predict the missing words based solely on contextual clues from surrounding text. Concurrently, it undertakes the *Next Sentence Prediction (NSP)* task, which assesses BERT’s ability to determine whether two consecutive sentences logically follow each other within a passage. This dual approach fosters a coherent understanding of sequence structure, crucial for contextual interpretation.

Upon completion of pre-training, BERT transitions into the fine-tuning phase, where it is tailored to additional specific downstream tasks such as sentiment analysis, question answering, and named entity recognition. During this fine-tuning, the model is trained on labeled data pertinent to these tasks, enabling it to leverage its pre-trained knowledge effectively and enhance its performance on targeted objectives. The pre-training and fine-tuning phases of the BERT model are depicted in Figure 2.8. In this process, the **[CLS]** token is used to represent the entire sequence for classification tasks, while the **[SEP]** token separates distinct segments, enabling the model to handle tasks that require understanding relationships between sentences. While each specific fine-tuning task is illustrated in Figure 2.9.

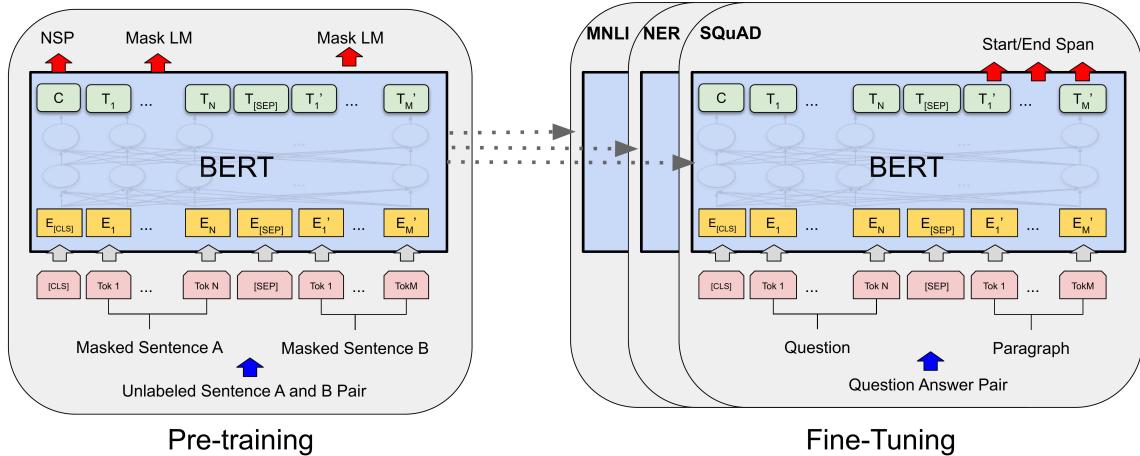


Figure 2.8: Overall pre-training and fine-tuning procedures for BERT

Building upon the BERT framework, *DistilBERT* [150] offers a more compact and efficient alternative that retains much of the original model’s performance while significantly reducing its size and inference time. By employing knowledge distillation [15] [61], DistilBERT compresses the knowledge of the larger BERT model, resulting in a model that is 60% faster and 40% smaller while retaining approximately 97% of its language understanding capabilities. Additional notable works are *RoBERTA* [110] and *ALBERT* [86].

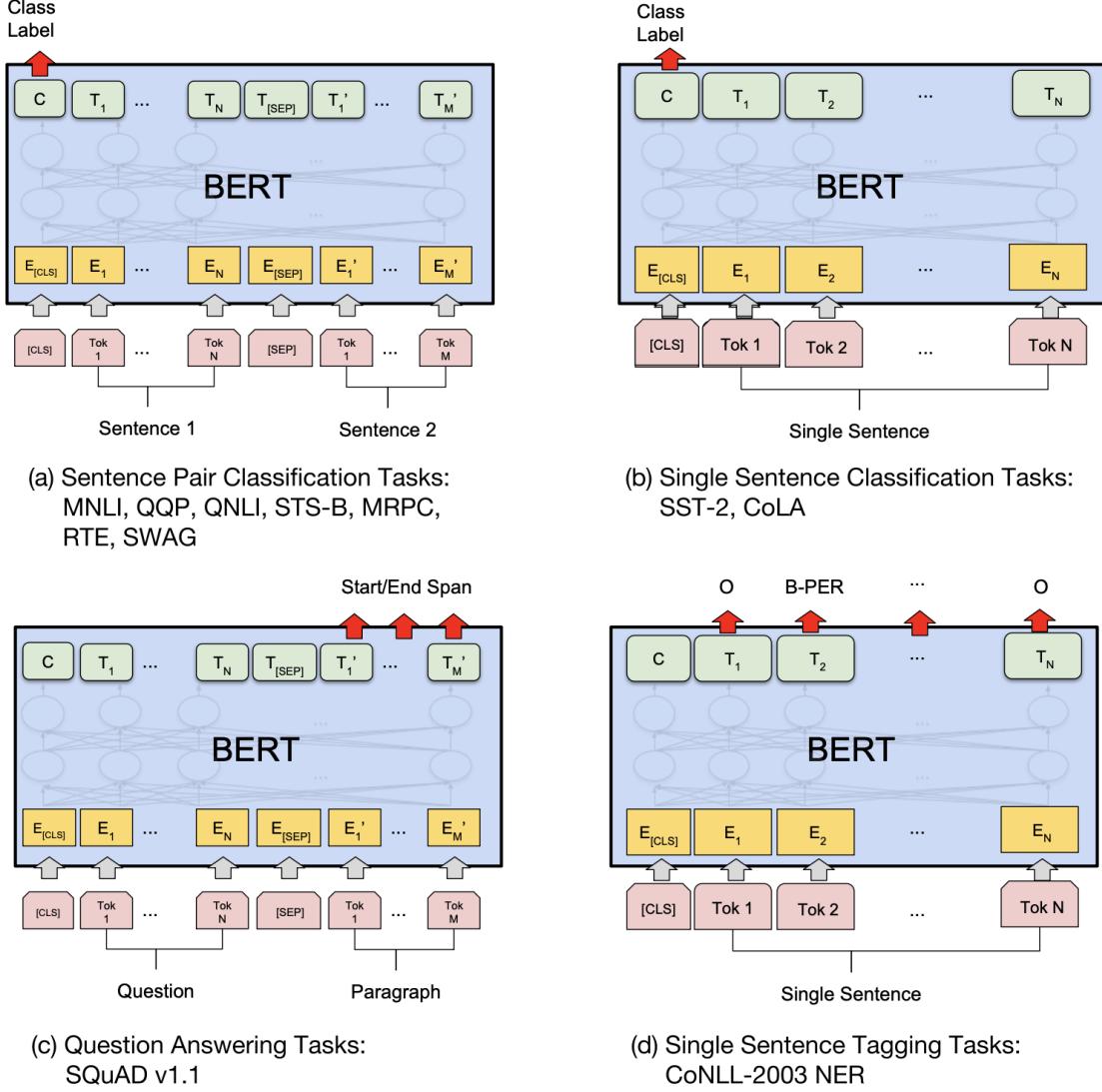


Figure 2.9: Illustrations of Fine-tuning BERT on Different Tasks.

2.1.3 Decoder-only Model: GPT-1 and GPT-2

Among the prominent decoder-only pre-trained language models (PLMs) are *GPT-1* [139] and *GPT-2* [140], both developed by OpenAI. These models serve as foundational pillars for the development of subsequent, more advanced Large Language Models (LLMs) like *GPT-3* [13] and *GPT-4* [131].

GPT-1 represents a significant milestone by demonstrating that a decoder-only Transformer model can achieve strong performance across a diverse array of natural language tasks. This is accomplished through *Generative Pre-Training (GPT)*, which utilizes a self-supervised learning approach focused on next word prediction across a varied corpus of unlabeled text. After this pre-training phase, the model undergoes discriminative fine-tuning tailored to specific downstream tasks, often requiring only a limited number of labeled samples. As illustrated in Figure 2.10, *GPT-1* sets the stage for future iterations, each enhancing the architectural design and improving performance on various language tasks. Building on the principles established by *GPT-1*, *GPT-2* further illustrates the capacity of language models to acquire the ability to perform specific natural language tasks without explicit supervision. This is achieved through training on the extensive *WebText* dataset [140], which comprises millions of web pages. *GPT-2* retains the overall architecture of its predecessor but introduces several key modifications: layer normalization is relocated to the input of each sub-block, additional normalization is applied after the final self-attention block, and adjustments are made to account for residual connections. Moreover, *GPT-2* increases the vocabulary size to 50257 and extends the context size from 512 to 1024 tokens, thus enhancing its capability to process and generate language.

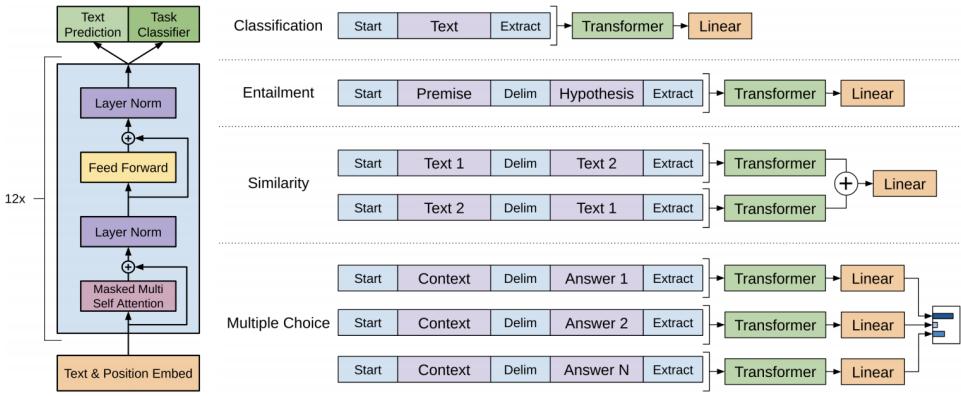


Figure 2.10: GPT-1 Transformer architecture and training objectives (left). Input transformations for fine-tuning on different tasks (right).

2.2 Large Language Models (LLMs)

The rapid advancement of Large Language Models (LLMs) marks the current, fourth wave in the evolution of language modeling, as we observe at the time of writing this thesis. Building upon the foundational work discussed in Section 2.1, these models, now equipped with **billions of parameters**, have consistently shown that increasing model size correlates with enhanced capabilities. This trend is further supported by the growth in the scale of training datasets, which now reach approximately more than 1 trillion tokens (an order of magnitude larger than previous models). For example, GPT-2 was trained on 10 billion tokens, while GPT-4 was trained on approximately 14 trillion, leading to significant improvements in performance [122].

A pivotal moment in the mainstream adoption of LLMs occurred in November 2022 with the release of *ChatGPT* [130]. This development allowed the general public to interact with these sophisticated models at minimal cost, thereby democratizing access to cutting-edge AI. As a result, it catalyzed a widespread perception that artificial intelligence had made a substantial leap forward, shifting the narrative from futuristic speculation to present-day reality.

Central to the success of LLMs is the self-attention mechanism, which enables these models to generate coherent and contextually relevant text by capturing long-range dependencies between words. By attending to the most relevant parts of the input sequence, LLMs are able to model complex linguistic structures, resulting in highly accurate and contextually appropriate outputs. This capacity for nuanced understanding and generation has made LLMs essential tools across a wide array of natural language processing tasks.

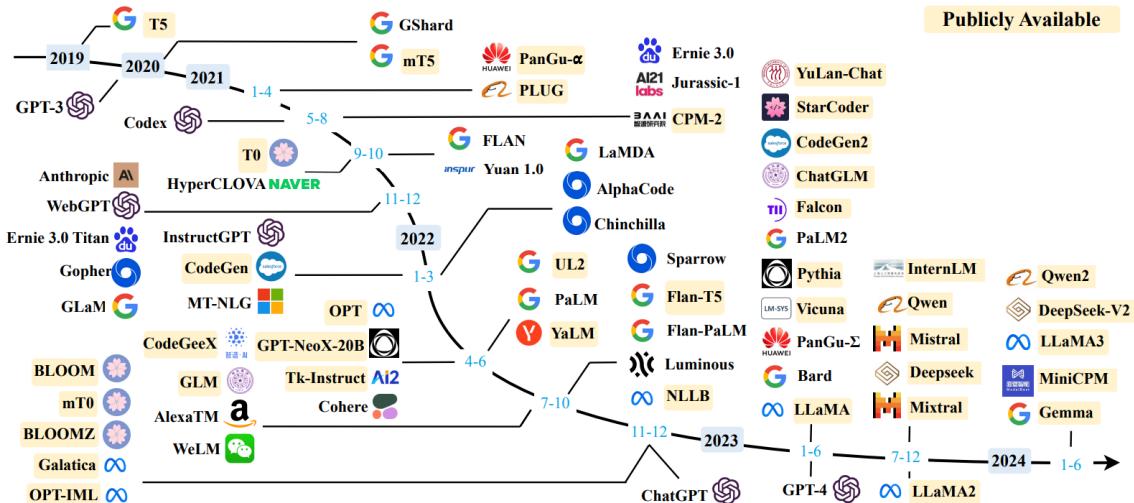


Figure 2.11: Infographic of Large Language Models releases (Source: [223])

2.2.1 Open source

While early developments in LLMs were predominantly driven by proprietary systems, such as OpenAI's GPT-3 [13] and Google's PaLM [30], the open-source community began to make significant strides only later. Initially, open-source alternatives like *Open Pre-trained Transformer (OPT)* [219], *BLOOM* [199], and *GPT-Neo* [11] lagged behind their commercial counterparts in terms of performance and scalability. However, this dynamic shifted with the release of Meta's *LLaMA 2* in 2023 [172], which demonstrated results comparable to those of proprietary models, marking a pivotal moment for open-source LLMs. This breakthrough not only closed the gap between closed and open systems but also catalyzed further innovation within the open-source ecosystem. Early open-source models like *OPT* and *BLOOM* provided foundational contributions to the development of Large Language Models, influencing subsequent architectures such as *LLaMA*, *Falcon* [1], and *MPT* [104]. Additionally, fine-tuned variants like *Alpaca* [167], *Vicuna* [28], and *Orca* [124], tailored to specific tasks and domains, have enhanced the flexibility and utility of these models. It's important to distinguish between models that are truly *open-source* and those with *open weights*. While "open-source" implies full transparency and accessibility to both the model's architecture and training data, models with "open weights" often make only the final trained parameters available. In some cases, licensing restrictions prevent full use, making the models technically not fully open. This difference is critical, as the absence of access to training data or limitations on use can hinder reproducibility and further development. The year 2023 has been widely recognized as the "year of open-source" in the context of LLMs [48], as the community saw a surge in the release of powerful, open models that began to rival proprietary solutions both in capability and accessibility, shifting the landscape of AI development.

2.2.2 Scaling laws

The optimization of LLMs poses significant challenges, particularly in determining how best to allocate computational resources to enhance performance. A key aspect of this optimization lies in understanding scaling laws, which provide a framework for analyzing the relationships between model size, training data, and computational cost. Empirical evidence suggests that LLMs demonstrate power-law behavior [77], wherein increasing model parameters or training data yields diminishing returns in performance, necessitating careful consideration of resource allocation during model development (Figure 2.12 and 2.13). Recent studies [77] indicate that as LLMs scale, they not only improve in language understanding and generation capabilities but also exhibit critical thresholds beyond which additional resources yield marginal gains. While larger models benefit from expanded datasets and increased parameters, these improvements plateau at certain points, highlighting the need for strategic investments in both architecture and data quality. Furthermore, innovations such as sparse architectures and efficient training methods challenge conventional scaling paradigms, suggesting that optimal performance may be achieved without direct linear scaling of computational costs. Understanding these scaling dynamics will be essential for creating more efficient models capable of delivering high-quality outputs in increasingly complex applications. By analyzing the interplay between model size, training data, and computational requirements, researchers and practitioners can make informed decisions that optimize performance while managing costs effectively.

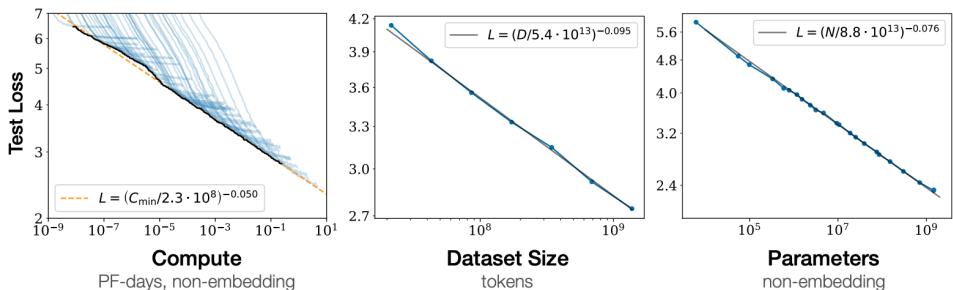


Figure 2.12: Language model performance increases steadily with larger model sizes, expanded dataset sizes, and greater computational resources during training. For optimal performance, it is crucial to scale all three components together. Empirical performance demonstrates a power-law relationship with each factor when not limited by the other two. (Source: [77])

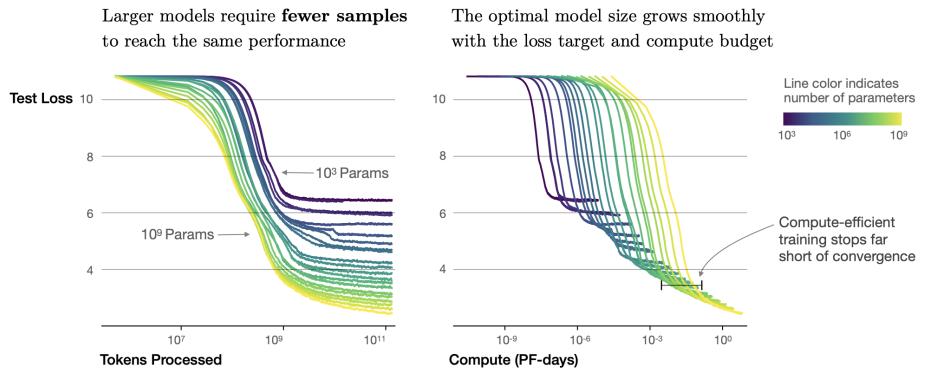


Figure 2.13: Series of language model training runs, with models ranging in size from 10^3 to 10^9 parameters (excluding embeddings). Source: [77]

2.3 Retrieval Augmented Generation (RAG)

LLMs have demonstrated impressive capabilities across a range of NLP tasks by leveraging vast amounts of training data. These models rely on their internal parameters as an "offline knowledge base" to generate text. However, this internal knowledge is inherently limited to the data seen during training, leading to a phenomenon known as *hallucination*, where the model produces factually incorrect or inconsistent information. This problem becomes more pronounced in knowledge-intensive tasks that require specific, up-to-date information, beyond the model's training data.

To address this limitation, *Lewis et al.* in 2020 introduced *Retrieval-Augmented Generation (RAG)* [91], a framework that integrates an external retrieval mechanism with generative models. RAG combines retrieval-based methods with generative capabilities to enhance factual accuracy and mitigate hallucination.

RAG is particularly effective for tasks like open-domain question answering, where precise and timely information is critical. The framework allows a generative model to retrieve relevant, up-to-date passages from a large external corpus during the generation process. By doing so, RAG enables the model to produce responses grounded in real-world data, significantly reducing the risk of factual errors.

The RAG architecture operates in a two-step pipeline:

1. **Retrieval step:** A retriever model, typically based on dense passage retrieval, searches a large corpus of documents to identify the most relevant passages for a given query.
2. **Generation step:** A generative model, such as GPT-4 or LLaMA 3.1, uses the retrieved passages to generate a coherent, contextually accurate response, combining the retrieved information with its internal knowledge.

By incorporating real-time retrieval, RAG effectively addresses the limitations of LLMs in handling dynamic and domain-specific knowledge, leading to more accurate and reliable outputs in knowledge-intensive tasks. However, the retrieval component of RAG introduces significant challenges, particularly in efficiently managing and searching through vast corpora of documents or knowledge bases. This is where **Vector Database Management Systems (VDBMS)** play a pivotal role, providing a robust infrastructure to support the retrieval needs of systems like RAG.

2.3.1 Vector Database Management Systems (VDBMS)

Vector Database Management Systems (VDBMS) [165] have emerged as a critical technology in the landscape of modern data management, driven by the rise of machine learning models and the increasing need to store and query high-dimensional vector embeddings. These embeddings are used to represent complex data points in a continuous, high-dimensional space and are integral to the retrieval process in RAG, where retrieving semantically relevant information is essential for grounding the generative model's output.

At the core of VDBMS is the efficient representation and management of these high-dimensional vectors, which correspond to data embeddings generated by machine learning models such as transformers or BERT. The process of converting data into smaller chunks, embedding them into vectors, and organizing these embeddings into a searchable structure is known as *indexing*. Therefore, vector databases allow the system to rapidly and accurately retrieve the most relevant information from an external corpus by calculating similarities between the query vector and stored vectors. This is crucial for mitigating the risk of hallucination in LLMs, as the real-time retrieval of relevant, up-to-date knowledge ensures that the generated responses are factually grounded.

Unlike traditional relational databases that store structured data in tables and use indexing techniques like *Hierarchical Navigable Small World (HNSW)* graphs [116], VDBMS focus on tasks like nearest neighbor search in high-dimensional spaces, which is essential for identifying vectors (representing documents or knowledge) that are closest to the query vector. To handle the computational complexity of these operations, VDBMS often rely on *Approximate Nearest Neighbor (ANN)* [2] or other search algorithms, which enable efficient querying and retrieval from large-scale vector spaces.

These spaces are typically populated with either *dense* or *sparse* vectors, depending on the nature of the data representation. Dense vectors, generated by deep learning models like word embeddings,

contain non-zero values in most dimensions and capture semantic relationships effectively in high-dimensional spaces. In contrast, sparse vectors, often produced by traditional methods like *TF-IDF* (*Term Frequency-Inverse Document Frequency*) [74], contain many zero elements and represent data more explicitly, though they may lack the depth of semantic information found in dense representations, as illustrated in Figure 2.14. However, both types of vectors are affected by the *curse of dimensionality*, where the performance of nearest neighbor search degrades as the number of dimensions increases, making efficient search algorithms like ANN crucial for VDBMS.

$$\mathbf{v}_{sparse} = \begin{bmatrix} 0 \\ 0 \\ 3 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{v}_{dense} = \begin{bmatrix} 2 \\ 5 \\ 3 \\ 1 \\ 1 \\ 4 \end{bmatrix}$$

(a) Sparse Vector
(b) Dense Vector

Figure 2.14: Sparse and Dense Vectors compared

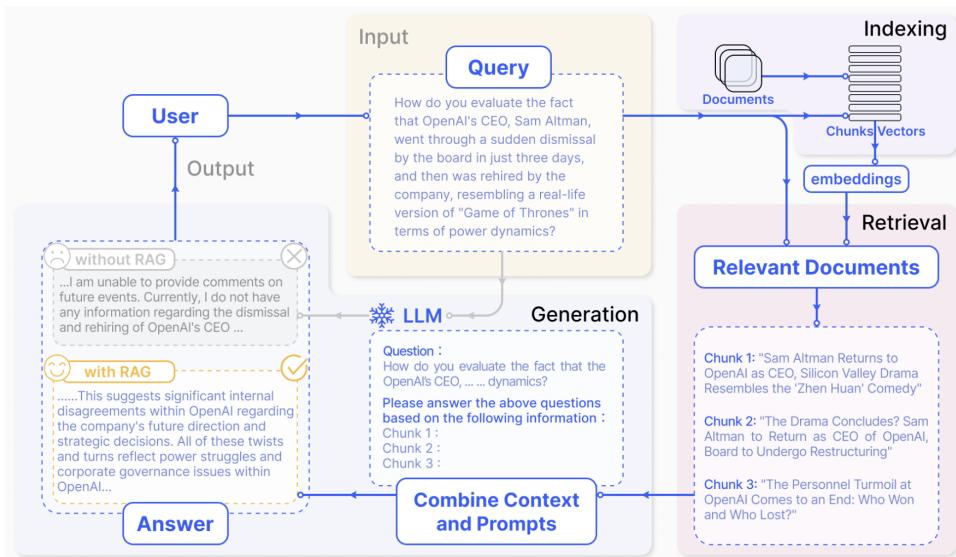


Figure 2.15: A representative instance of the RAG process applied to question answering. It mainly consists of 3 steps. 1) Indexing. Documents are split into chunks, encoded into vectors, and stored in a vector database. 2) Retrieval. Retrieve the Top k chunks most relevant to the question based on semantic similarity. 3) Generation. Input the original question and the retrieved chunks together into LLM to generate the final answer. (Source: [49])

2.3.2 Rerankers

Following the initial retrieval of relevant documents from a vector database using ANN or other searching algorithms, rerankers play an essential role in enhancing the relevance of the retrieved information [18]. A reranker is a critical component in the information retrieval ecosystem that evaluates and reorders search results to better align them with a specific query. Specifically, rerankers improve search quality by assessing the semantic relevance between documents and queries, refining the initial set of candidates retrieved from the vector database.

They can be categorized into two main types: **score-based** rerankers and **neural network-based** rerankers. Score-based rerankers aggregate candidate lists from various sources and apply scoring mechanisms, such as weighted scoring or *Reciprocal Rank Fusion (RRF)* [33], to produce a unified and prioritized list based on relevance. This approach is efficient and is commonly employed in traditional search systems due to its lightweight nature. Conversely, neural network-based rerankers,

often referred to as *cross-encoder* rerankers, utilize deep learning models to analyze the relevance between a query and candidate documents. This method allows for a more nuanced understanding of semantic proximity, resulting in a refined ordering of the retrieved results.

Integrating a reranker into a RAG application enhances the precision of generated responses by narrowing the context to a curated set of highly relevant documents, thereby enabling LLMs to produce higher-quality answers. However, this integration introduces trade-offs related to increased computational costs and latency, necessitating a careful assessment of the application’s specific needs to determine the appropriateness of including a reranker.

2.3.3 RAG paradigms

In their study, *Gato et al.* [49] conducted an in-depth investigation into the emerging paradigms within the domain of RAG. As depicted in Figure 2.16, they identified three primary architectures: *Naive RAG*, *Advanced RAG*, and *Modular RAG*.

Naive RAG represents the classic process for Retrieval-Augmented Generation, encompassing three fundamental steps. First, *Indexing* involves partitioning the document corpus into smaller segments and constructing a vector index using an encoder. Second, during *Retrieval*, relevant document fragments are retrieved based on the similarity between the input query and the indexed chunks. Finally, the *Generation* phase produces an answer to the query conditioned on the retrieved context.

Advanced RAG was introduced to address several limitations inherent in Naive RAG, particularly in retrieval, generation, and context augmentation. This paradigm incorporates additional steps both before and after the retrieval process. *Pre-retrieval* techniques such as query rewriting, query routing, and semantic expansion are employed to better align the query with the document chunks. *Post-retrieval*, strategies such as re-ranking the retrieved documents help mitigate issues like the “*Lost in the Middle*” phenomenon, where critical information might be overlooked due to its position. Additionally, context filtering and compression methods are used to reduce the input window length, improving overall efficiency.

Modular RAG represents a more flexible and sophisticated evolution of the RAG framework, transcending the conventional Retrieval-Generation pipeline. It introduces specialized functional modules, such as query search engines and mechanisms for fusing multiple retrieved answers. Technologically, it integrates advanced techniques like retrieval fine-tuning, reinforcement learning, and other optimization strategies. Modular RAG is characterized by its orchestration of different modules, allowing for varied patterns of retrieval and generation. Conceptually, it builds upon the previous paradigms, with Advanced RAG being a particular instance of Modular RAG, and Naive RAG serving as a simplified form of Advanced RAG.

It is important to note that these paradigms are closely interconnected, with each one building on and refining the capabilities of the previous. Rather than being separate approaches, they represent a continuous evolution in the development of retrieval-augmented generation techniques.

2.3.4 RAG vs Fine-Tuning

Beyond Retrieval-Augmented Generation, another prominent approach for enhancing Large Language Models is fine-tuning (FT). Unlike RAG, which retrieves external knowledge at inference time to incorporate real-time information, FT modifies the internal parameters of the model by training it on specific datasets. This enables the model to internalize new knowledge patterns, making it well-suited for tasks that require replicating particular styles, structures, or domain-specific behaviors [49, 68].

One of the key distinctions between the two methods lies in their adaptability. FT requires re-training whenever updates or new knowledge are needed, making it computationally expensive and less flexible for rapidly evolving data landscapes. Conversely, RAG can dynamically access and integrate up-to-date information without retraining, though this comes with higher latency and ethical considerations surrounding real-time data retrieval. FT, while offering deep customization, struggles with new or unfamiliar data unless retrained, making it less effective for applications where knowledge changes frequently [49].

Recent evaluations [132] highlight RAG’s superiority over unsupervised FT, especially in tasks that require the handling of new factual information. Fine-tuning often fails to equip models with such knowledge unless exposed to multiple variations during training. However, RAG and FT can

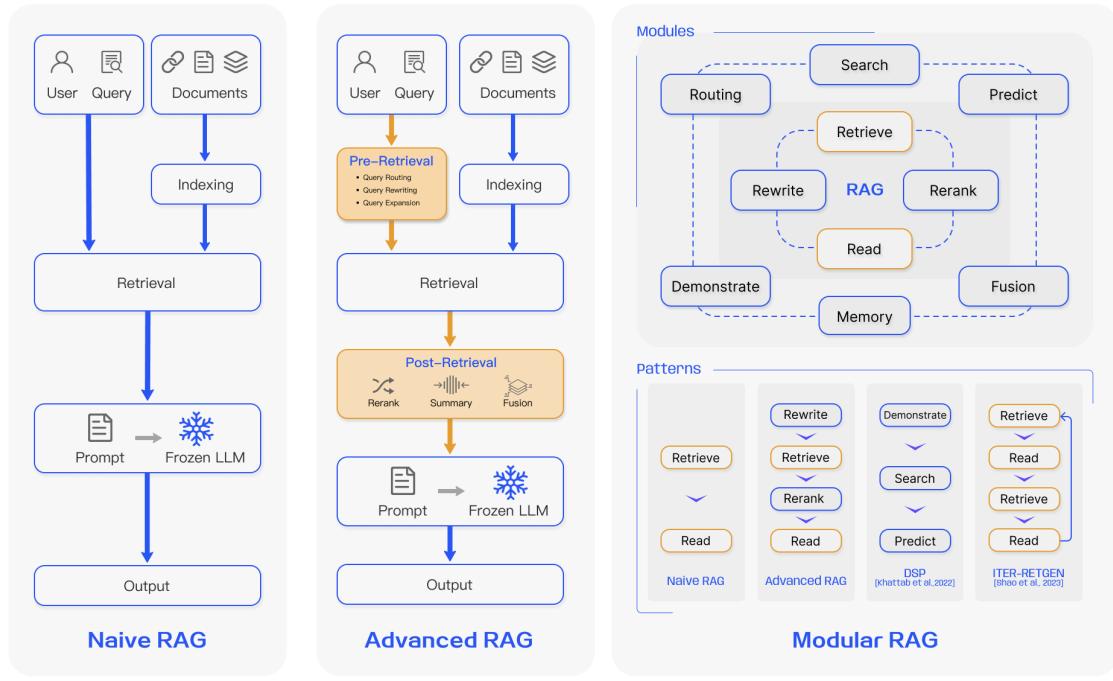


Figure 2.16: Comparison between the three paradigms of RAG. Source: [49]

complement each other, with RAG providing real-time adaptability and FT offering granular control over model behavior. Their combined use may achieve optimal performance in applications requiring both dynamic knowledge access and deep task-specific adjustments [132, 49]. Figure 2.17 illustrates the best-suited techniques for different requirements in terms of external knowledge and model adaptation.

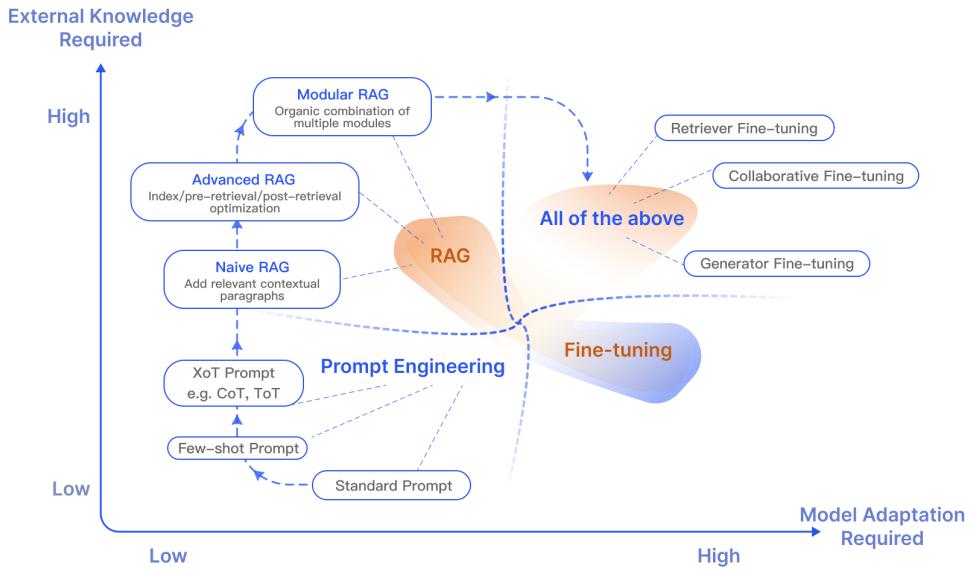


Figure 2.17: RAG compared with other model optimization methods in the aspects of “External Knowledge Required” and “Model Adaption Required”. Source: [49]

2.3.5 Prompt Engineering

As LLMs become increasingly integrated into various applications, addressing their inherent limitations—such as susceptibility to generating irrelevant or incorrect outputs—has emerged as a critical challenge. Prompt engineering serves as an essential strategy to overcome these limitations by op-

timizing the input prompts used to elicit desired responses from LLMs, ultimately enhancing their performance across a range of tasks [149]. This subsection explores key contributions and techniques in prompt engineering, illustrating its significance in maximizing the utility of LLMs.

In scenarios where new tasks arise without extensive training data, *zero-shot* and *few-shot* prompting have proven to be effective methodologies. Zero-shot prompting, as demonstrated by *Radford et al.* [140], involves presenting a task description to the model without providing any specific examples, relying solely on its pre-trained knowledge. Conversely, few-shot prompting, introduced by *Brown et al.* [13], enhances model performance by incorporating a limited number of annotated examples within the prompt, enabling the model to better understand task-specific nuances.

For reasoning and logic tasks, *Chain-of-Thought (CoT)* prompting, proposed by *Wei et al.* [197], encourages models to articulate their thought processes, thus improving their reasoning capabilities. Various techniques extend the principles of CoT prompting to facilitate more nuanced reasoning in complex tasks, such as: *Auto-Cot* [222], *Self-Consistency* [189], *Chain of Table Prompting* [196] and many others.

To mitigate the issue of hallucinations—instances where models generate incorrect or misleading information—techniques such as Retrieval-Augmented Generation [91], *ReAct* [208], and other chained methods like *Chain-of-Verification (CoVe)* [38], *Chain-of-Note (CoN)* [211], and *Chain-of-Knowledge (CoK)* [100] have been developed. These approaches leverage external knowledge sources or structured reasoning paths to enhance the accuracy of model outputs. *Active Prompting*, introduced by *Diao et al.* [39], enhances Large Language Models' performance on complex reasoning tasks by dynamically selecting the most impactful questions for annotation through uncertainty-based active learning. This innovative approach outperforms traditional chain-of-thought methods.

When focusing on fine-tuning and optimization, several advanced techniques come into play. *Zhou et al.* introduced *Automatic Prompt Engineer (APE)* [229], a method that dynamically generates effective prompts based on user input. To bridge the reasoning gap, *Paranjape et al.* introduced *Automatic Reasoning and Tool-use (ART)* [136], enabling LLMs to tackle complex problems by integrating external tools and automating multi-step reasoning. *Chia et al.* [26] proposed *Contrastive Chain-of-Thought (CCoT)* prompting, which enhances learning by providing both valid and invalid reasoning examples, leading to improved performance in strategic evaluations. Addressing emotional intelligence, *Emotion Prompting* by *Li et al.* [93] incorporates emotional stimuli into prompts, significantly enhancing LLM performance across tasks. For code generation, techniques such as *Scratchpad Prompting*, introduced by *Nye et al.* [128], facilitate the generation of intermediate reasoning steps, while *Chen et al.* presented *Program of Thoughts (PoT)* [23] prompting to improve numerical reasoning by leveraging external interpreters. Additionally, *Structured Chain-of-Thought (SCoT)* prompting by *Li et al.* [95] incorporates program structures for more effective code generation, and *Chain-of-Code (CoC)* prompting by *Li et al.* [94] improves reasoning tasks through the use of pseudocode. To enhance understanding of user intent, *Rephrase and Respond (RaR)* prompting, introduced by *Deng et al.* [36], allows LLMs to reformulate and elaborate on user queries. This approach not only improves response accuracy but also clarifies semantic ambiguities inherent in casual human queries. Building on the need for improved reasoning strategies, *Take a Step Back Prompting*, introduced by *Zheng et al.* [224], enables advanced language models like *PaLM-2L* to extract high-level concepts through a two-step process of abstraction and reasoning. This technique specifically enhances LLMs' performance in intricate tasks such as *STEM (science, technology, engineering and mathematics)* problem-solving and multi-hop reasoning. Collectively, these innovations mark substantial progress in the fine-tuning and optimization of LLMs, with most methods emerging in 2023. Figure 2.18 provides a taxonomy of these prompt engineering techniques, as outlined in *Sahoo's survey* [149], illustrating the breadth of recent advancements. As a result, this surge in research demonstrates a concerted effort to maximize the capabilities of LLMs, yet it is evident that we are still at the nascent stage of this evolving field.

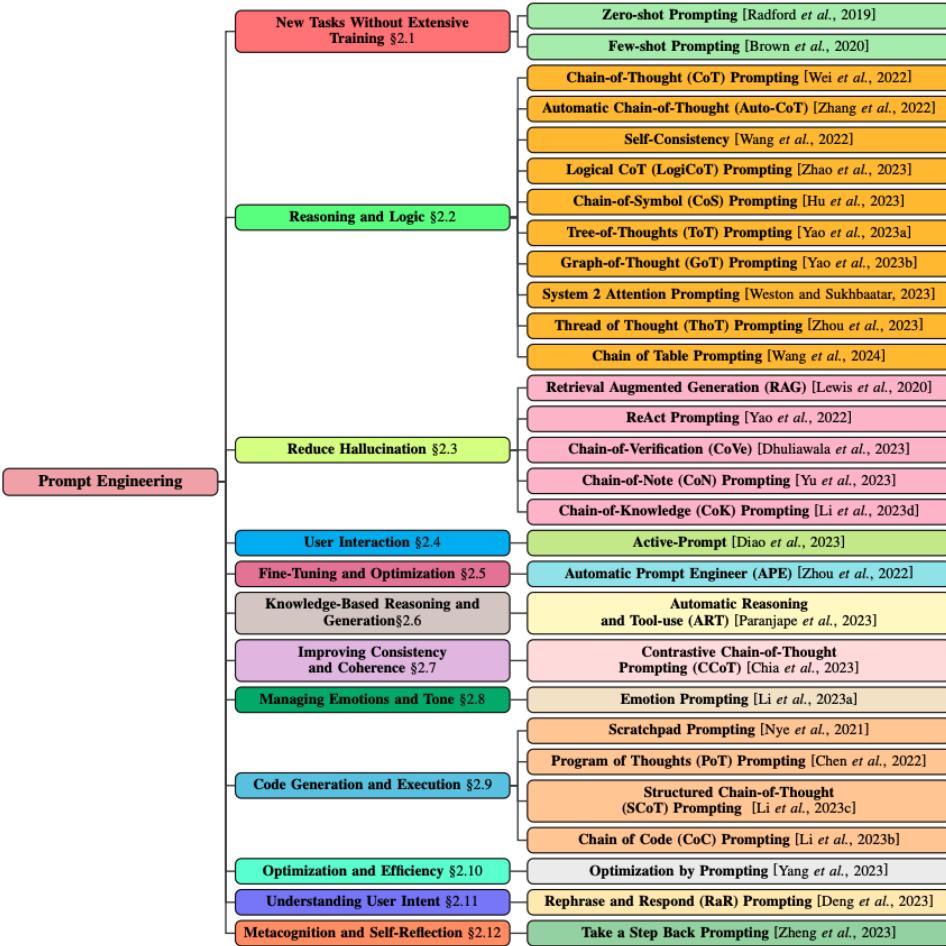


Figure 2.18: Taxonomy of prompt engineering techniques in LLMs, organized around application domains, providing a nuanced framework for customizing prompts across diverse contexts. Source [149]

2.4 LLM Evaluation

The evaluation of Large Language Models presents a unique set of challenges that extend beyond traditional Natural Language Processing assessment methodologies. As these models continue to evolve in complexity and capability, so too must our approaches to measuring their performance and understanding their limitations. This section provides a comprehensive overview of LLM evaluation techniques, beginning with an examination of traditional NLP evaluation metrics and tasks. We will then explore the critical distinctions between automatic and human evaluation methodologies, each offering valuable insights into different aspects of model performance. Additionally, we will examine the development and application of LLM-specific benchmarks, which have emerged as essential tools for comparing model capabilities across a wide range of tasks and domains. These benchmarks serve as standardized measures of progress in the field, allowing researchers and practitioners to assess advancements in LLM technology systematically. Finally, we will investigate the emerging paradigm of using LLMs themselves as judges in evaluation processes. This meta-evaluative approach raises intriguing questions about the nature of assessment and the potential for models to provide nuanced, context-aware evaluations of language generation and understanding tasks. Through this exploration, we aim to provide a nuanced understanding of the current state of LLM evaluation, its challenges, and the directions in which this crucial area of research is evolving.

2.4.1 Traditional Natural Language Processing (NLP) evaluation

As outlined in Section 2.1, the primary motivation for developing Large Language Models has been to enhance performance across various natural language processing tasks, which encompass both language understanding and generation. In response to this objective, an extensive body of research

has emerged, focusing on evaluating the capabilities of LLMs in diverse NLP tasks.

NLP is composed of several subfields, each characterized by specific tasks and evaluation metrics. According to *Chang et al.* [16], these subfields include *Natural Language Understanding (NLU)*, *Reasoning*, *Natural Language Generation (NLG)*, *Multilingual Processing*, and *Factuality*.

Natural Language Understanding (NLU) serves as a fundamental area within NLP, focusing on the interpretation of human language through various tasks. NLU can be further divided into three primary tasks: **Sentiment Analysis**, **Text Classification**, and **Natural Language Inference (NLI)**.

The task of **Text Classification** involves the assignment of predefined labels to text data, spanning a broad spectrum of categories that extend beyond mere sentiment. For instance, *Liang et al.* [102] demonstrated that the *GLM-130B* model achieved a remarkable accuracy of 85.8% in a range of miscellaneous text classification tasks, thereby illustrating its effectiveness across diverse textual contexts.

Within the scope of text classification, **Sentiment Analysis** specifically focuses on discerning the emotional tone of a text, categorizing it into classes such as *positive*, *negative*, or *neutral*. Performance in sentiment analysis is commonly assessed using metrics such as *accuracy*, *F1-score*, and *precision*. Recent studies conducted by *Liang et al.* [102] and *Zeng et al.* [217] indicate that LLMs demonstrate strong capabilities in sentiment analysis tasks, particularly for well-resourced languages. However, there remains a critical need for enhanced emotional understanding in under-resourced languages, highlighting an important area for future research.

Furthermore, **Natural Language Inference (NLI)** evaluates the logical relationship between a given premise and hypothesis, determining whether the hypothesis logically follows from the premise. Notably, *Qin et al.* [138] found that *ChatGPT* outperformed *GPT-3.5* in NLI tasks, underscoring significant advancements in logical reasoning capabilities. Nevertheless, as pointed out by *Lee et al.* [89], LLMs continue to encounter difficulties in accurately representing human disagreement, indicating substantial opportunities for further enhancement in NLI tasks.

In addition to NLU, **Reasoning** tasks necessitate that models engage in logical deduction and problem-solving. These tasks frequently utilize metrics such as *accuracy* to evaluate a model's proficiency in generating correct answers based on the premises provided. A prominent example of reasoning tasks in natural language is multi-hop question answering, where models must integrate information from multiple text sources to arrive at an accurate conclusion.

Transitioning to the domain of **Natural Language Generation (NLG)**, particularly in tasks involving summarization and question answering, models are evaluated using metrics such as *ROUGE* [103] and *BLEU* [135]. The *ROUGE* score measures the overlap between generated summaries and reference summaries, whereas *BLEU* is employed in machine translation tasks to assess the fidelity of generated text against a reference translation. Recent advancements in LLMs have yielded promising results in producing coherent and contextually relevant outputs, as illustrated by *Zheng et al.* [226], who curated a comprehensive dataset for LLM conversations titled *LMSYS-Chat-1M*.

The subfield of **Multilingual Processing** emphasizes the capability of models to comprehend and generate text in multiple languages. Evaluation of this task typically employs metrics such as *cross-lingual accuracy* and *language coverage*, which assess a model's performance across a variety of linguistic contexts. As the use of multilingual models becomes increasingly widespread, research efforts are directed toward enhancing their effectiveness in low-resource languages.

Finally, the assessment of **Factuality** ensures that the content generated aligns with verified information. Metrics such as *factual accuracy* and *fact-checking scores* are employed to evaluate the reliability of the outputs. Studies conducted by *Wang et al.* [182] underscore the ongoing challenge of ensuring factual correctness within LLMs, emphasizing the need for continuous refinement of evaluation methodologies. Additionally, the common metrics used for NLP evaluation are presented in Table 2.2, while the corresponding formulas for these metrics can be found in Table 2.3.

Table 2.2: Common Metrics for NLP Evaluation

Metric	Use Case	Pros	Cons
Accuracy	Text classification tasks (e.g., sentiment analysis, topic categorization)	Simple to compute and interpret; widely used in binary/multi-class classification	Can be misleading for imbalanced datasets where the majority class dominates
Precision	Binary and multi-class classification tasks, especially when false positives are costly (e.g., spam detection)	Effective when false positives need minimization	Does not account for false negatives, which can be problematic in certain tasks
Recall	Binary and multi-class classification tasks, especially when false negatives are costly (e.g., medical diagnosis)	Good for detecting rare but important events	May yield many false positives if used alone
F1-Score	Class imbalance scenarios (e.g., fraud detection, named entity recognition)	Balances precision and recall, providing a single score	Does not differentiate between false positives and false negatives
BLEU Score	Machine translation and text generation tasks	Measures how closely generated text matches a reference translation	Sensitive to exact n-gram matches, which can penalize semantically correct but lexically different outputs
ROUGE Score	Text summarization and translation evaluation	Focuses on recall (important for summarization), and allows multiple reference translations	Can overvalue recall at the expense of precision; might not align with human judgment
Perplexity	Language model evaluation (e.g., GPT, BERT)	Measures how well a model predicts a sample and indicates model uncertainty	Can be difficult to interpret, especially across different models, and does not always correlate with human evaluation
Word Error Rate (WER)	Speech recognition and transcription tasks	Measures the error rate in speech-to-text systems, directly interpretable as a proportion of errors	Sensitive to small changes in phrasing; does not account for semantic correctness
Meteor	Machine translation and text generation tasks	Considers synonyms and stemming, which improves semantic evaluation compared to BLEU	More computationally intensive and less widely adopted compared to BLEU

Table 2.3: Common NLP Metrics and Their Formulas

Metric	Formula
Accuracy	$\frac{\text{Number of correct predictions}}{\text{Total number of predictions}} = \frac{TP+TN}{TP+TN+FP+FN}$
Precision	$\frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Positives (FP)}}$
Recall	$\frac{\text{True Positives (TP)}}{\text{True Positives (TP)} + \text{False Negatives (FN)}}$
F1-Score	$2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$
BLEU Score	$\exp \left(\min \left(1 - \frac{\text{Length of reference}}{\text{Length of candidate}}, 0 \right) \right) \prod_{n=1}^N (\text{Precision}_n)^{w_n}$
ROUGE Score	$\frac{\sum_{\text{match} \in \text{hypothesis}} \text{Count}(\text{match})}{\sum_{\text{n-gram} \in \text{reference}} \text{Count}(\text{n-gram})}$
Perplexity	$2^{-\frac{1}{N} \sum_{i=1}^N \log_2 P(w_i)}$
Word Error Rate (WER)	$\frac{\text{Substitutions} + \text{Insertions} + \text{Deletions}}{\text{Total words spoken}}$
METEOR¹	$10 \times \frac{\text{Precision} \times \text{Recall}}{\alpha \times \text{Precision} + (1-\alpha) \times \text{Recall}}$

2.4.2 Types of Evaluation

The evaluation of Large Language Models can be categorized into two primary types: **automatic evaluation** and **human evaluation**.

Automatic evaluation

Automatic evaluation, the more prevalent method, minimizes the need for intensive human participation, thereby facilitating a standardized evaluation process. *Lin and Chen* [107] proposed *LLM-EVAL*, a comprehensive automatic evaluation framework designed to assess open-domain conversations with LLMs across multiple dimensions. While, *Jain et al.* [66] streamlined model evaluation in real-world scenarios by removing the need for extensive labeling of new data, thereby improving efficiency.

Specifically, tailored benchmarks for automatic evaluation are discussed in Section 2.4.3.

When computational resources allow for extensive experimentation, the literature identifies key metrics for evaluating various aspects of LLMs, as summarized in Table 2.4.

Table 2.4: Key Metrics for Automatic Evaluation according to [16]

General Metrics	Metrics
Accuracy	Exact Match, Quasi-Exact Match, F1 Score, ROUGE Score
Calibration	Expected Calibration Error [53], Area Under the Curve [50]
Fairness	Demographic Parity Difference [216], Equalized Odds Difference [55]
Robustness	Attack Success Rate [178], Performance Drop Rate [230]

However, **accuracy** can be quantitatively assessed using metrics such as *Exact Match (EM)*, which determines whether the predicted output aligns with the expected result, alongside F1 Score and ROUGE Score.

¹In the METEOR metric, α is typically set to 0.9 to prioritize precision over recall.

Calibration metrics assess the alignment between the model’s confidence levels and actual prediction accuracy. One widely utilized metric is *Expected Calibration Error (ECE)*, which quantifies the calibration performance of models. *Tian et al.* [171] employed ECE to analyze the calibration of *Reinforcement Learning from Human Feedback (RLHF)* LMs, including *ChatGPT*, *GPT-4*, *Claude 1*, *Claude 2*, and *Llama 2*. To calculate ECE, model predictions are categorized based on confidence levels, and the average accuracy of predictions is computed within each confidence interval. Additionally, the *Area Under the Curve (AUC)* of selective accuracy and coverage serves as another critical metric for evaluating model performance [50].

Fairness is a critical aspect of model evaluation, focusing on the equitable treatment of various demographic groups, including attributes such as gender, race, and age. *DecodingTrust* [179] employs two key metrics to assess fairness: the *Demographic Parity Difference (DPD)*, which evaluates the equitable distribution of model predictions across different population groups, indicating significant prediction disparities and potential model bias when high; and the *Equalized Odds Difference (EOD)*, which aims to ensure that the model maintains similar error rates across diverse populations, thereby promoting fairness in predictive performance. Finally, **Robustness** assesses the model’s performance against challenging inputs, including adversarial attacks, shifts in data distribution, and noise. The *Attack Success Rate (ASR)* is crucial for evaluating the adversarial robustness of LLMs [186], while the *Performance Drop Rate (PDR)* is a newly developed metric designed to effectively assess the robustness of prompts in LLMs [230].

In scenarios where automatic evaluation methods are inadequate, **human evaluation** emerges as a necessary alternative. This is especially pertinent in open-generation tasks, where embedded similarity metrics, such as *BERTScore* [220], may fail to capture the qualitative aspects of generated content [127]. Although some generation tasks can incorporate automatic evaluation protocols, human assessment is often favored as it allows for a more nuanced evaluation, recognizing that model outputs can exceed traditional benchmark answers.

Human evaluation

Human evaluation, rather unsurprisingly, entails the assessment of model-generated results through the involvement of human evaluators. This method provides insights that are more closely aligned with real-world applications, yielding comprehensive and nuanced feedback. Typically, evaluators—ranging from experts and researchers to ordinary users—are invited to assess the outputs produced by LLMs. For instance, *Ziems et al.* [231] utilized expert annotations in their evaluation of generated content. Similarly, *Liang et al.* [102] employed human evaluation to analyze summarization and disinformation scenarios across six different models, while *Bang et al.* [6] focused on evaluating analogical reasoning tasks. In a notable study, *Bubeck et al.* [14] conducted a series of human-crafted tests with *GPT-4*, finding that its performance approached or even surpassed human capabilities in multiple tasks. Such evaluations necessitate that human assessors engage in comparative testing of model performances rather than relying solely on automated metrics.

It is important to acknowledge that human evaluations can exhibit high variance and instability, which may arise from individual and cultural differences [137]. Consequently, a combined approach that considers both human and automatic evaluation methods is often employed, taking into account the specific context of the evaluation. Table 2.5 provides an overview of the essential aspects of human evaluation, including the number of evaluators, evaluation criteria, and the evaluators’ expertise level.

Table 2.5: Summary of Key Factors in Human Evaluation according to [16]

Evaluation Criteria	Key Factor
Number of evaluators	Adequate representation [7], Statistical significance
Evaluation criteria	Accuracy [158], Relevance [228], Fluency [175], Transparency, Safety [69], Human alignment
Evaluator’s expertise level	Relevant domain expertise [129], Task familiarity, Methodological training

2.4.3 LLM Benchmarks

Given the sheer number of benchmarks available, this subsection will highlight a selection of them to provide a representative overview of the most prominent and widely-used benchmarks currently available. These benchmarks primarily assess the textual capabilities of LLMs across various categories. **Reasoning and Commonsense** benchmarks evaluate an LLM’s ability to apply logic and everyday knowledge to solve problems. **Language Understanding and Question Answering (QA)** tasks assess the model’s skill in interpreting text and providing accurate answers. In the **Coding** category, benchmarks measure the LLM’s capability to interpret and generate code. **Conversation and Chatbot** benchmarks evaluate the model’s proficiency in engaging in dialogue and providing coherent, contextually relevant responses. **Translation** benchmarks test the model’s accuracy in translating text between languages. **Math** benchmarks focus on a model’s ability to solve problems ranging from basic arithmetic to advanced topics like calculus. **Logic** benchmarks assess the model’s inductive and deductive reasoning abilities, while **Standardized Test** benchmarks utilize exams like the *SAT* or *ACT* to evaluate academic performance across various domains. Figure 2.19 illustrates the most commonly encountered benchmarks utilized in comparisons of recent LLM evaluations.

To specify, automatic benchmarks operate by performing a series of predefined tests on LLMs, as illustrated in Figure 2.20, where each test evaluates specific capabilities, and a scoring system assigns a numerical score based on the model’s performance across these tasks.

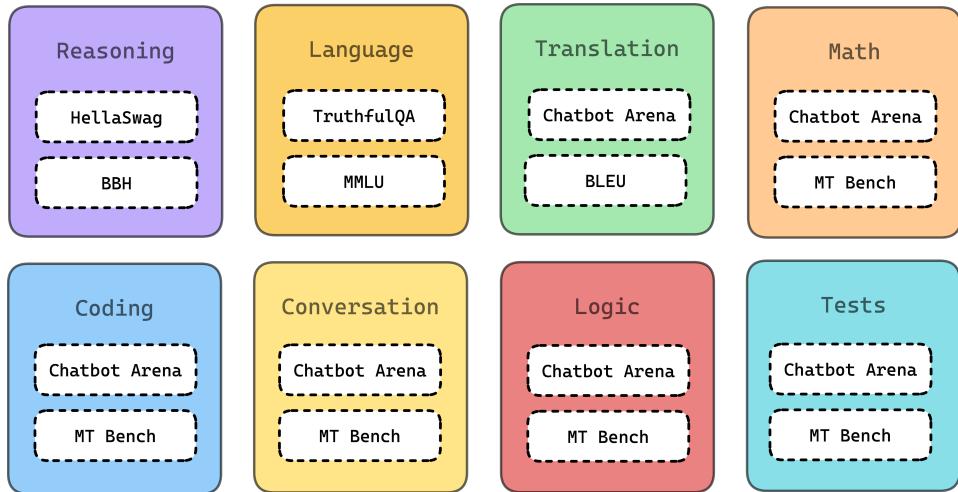


Figure 2.19: LLM Benchmark Categories (Source: [177])

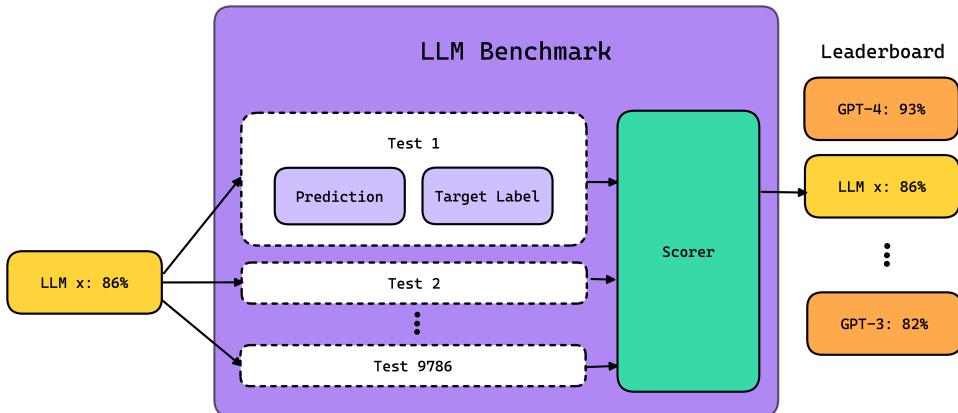


Figure 2.20: An LLM Benchmark Architecture (Source: [177])

HellaSwag is a benchmark designed to evaluate an LLM’s ability to understand commonsense reasoning by presenting models with a context and requiring them to select the most plausible continuation from multiple options; it comprises approximately 70,000 examples [214].

TruthfulQA assesses the capacity of language models to provide truthful answers to questions across a wide range of topics, consisting of 817 diverse questions across 37 categories, that are designed to probe factual knowledge [105].

MMLU (*Massive Multitask Language Understanding*) evaluates model performance on a wide array of tasks, such as reading comprehension, common sense reasoning, and factual knowledge, using a dataset of 57 tasks and over 2 million examples [59]. Additionally, there is a Pro version [193] that further refines and extends the original benchmark by focusing on more complex reasoning tasks and providing a deeper evaluation of language models’ capabilities.

MT-Bench [225] evaluates the quality of chat assistants by presenting them with a series of multi-turn open-ended questions, utilizing Large Language Models (LLMs) as judges (reference to Subsection 2.4.4).

BIG-Bench Hard (BBH) is a benchmark consisting of 23 challenging tasks selected from the original BIG-Bench suite, designed to assess tasks where language models previously struggled to outperform average human ratings [164]. This benchmark highlights the limitations of current models, as many BBH tasks necessitate multi-step reasoning. Notably, employing chain-of-thought (CoT) prompting significantly enhances model performance, allowing models like *PaLM* and *code-davinci-002* to exceed human performance on several tasks.

HumanEval evaluates the ability of models to generate correct code snippets based on specific problem statements, featuring 164 Python programming tasks, each accompanied by a prompt and expected output [20].

Chatbot Arena serves as an open platform for the evaluation and ranking of language models, leveraging feedback from more than 200,000 human voters. Unlike conventional benchmarks that rely on fixed metrics for scoring, this platform allows users to anonymously assess and compare pairs of AI models, such as *ChatGPT* and *Claude*, while remaining unaware of the models’ identities. Votes are counted towards the overall rankings only if the identities of the models remain hidden, with the final score reflecting the number of “upvotes” received. This methodology ensures a diverse assessment of conversational agents across a variety of dialogue scenarios [27]. An example of the tool in action is presented in Figure 2.21.

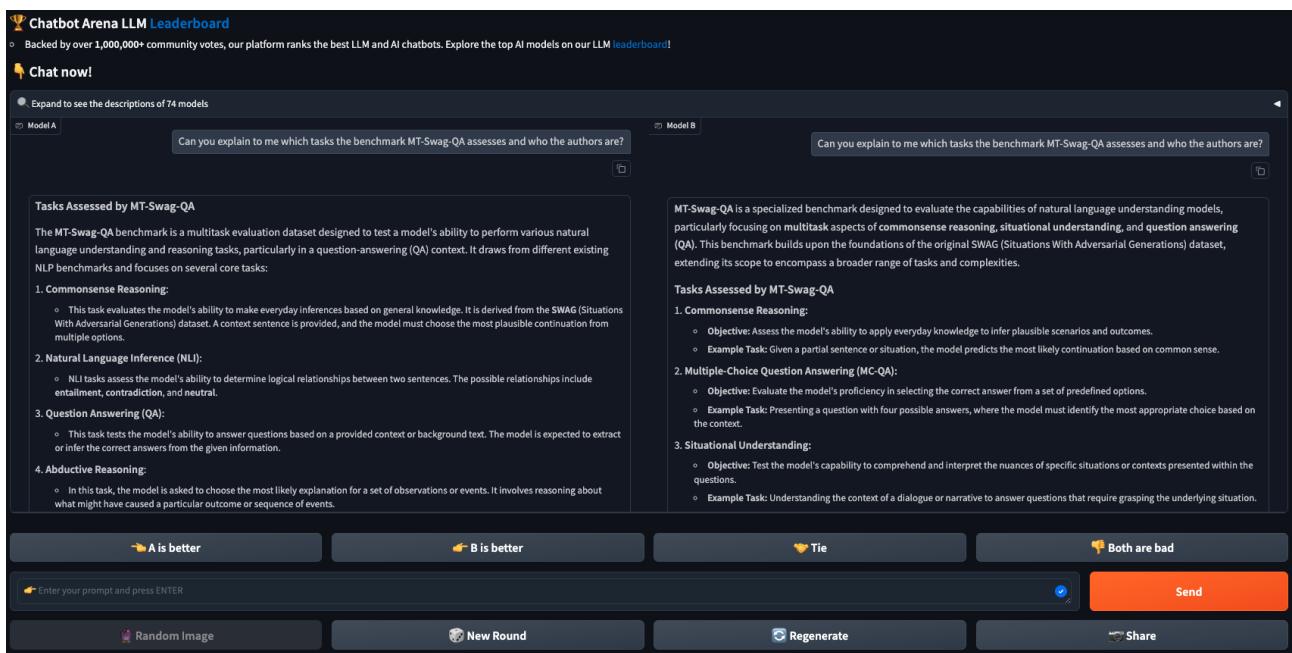


Figure 2.21: A clear example of LLM hallucination while interacting with Chatbot Arena website.

It is important to note with the advent of *multimodal* LLMs, new benchmarks are emerging to assess capabilities across multiple modalities. For a more thorough exploration of these developments, the recent work of *Jian Li et al.* [96] provides a valuable resource for investigating benchmarks tailored to multimodal systems.

2.4.4 LLM as Judge paradigm

Towards the end of 2023, a novel paradigm emerged within the landscape of Large Language Model evaluation: the **LLM-as-a-Judge** approach, introduced by *Zeng et al.* [225]. This methodology explores the potential of using one LLM to serve as an evaluator or "judge" for the outputs of other LLMs, particularly in the context of open-ended question assessments.

The authors identified several **biases** that may arise when LLMs are used in this evaluative capacity, including *position bias*, *verbosity bias*, and *self-enhancement bias*. Their contribution to the field includes the development of evaluation platforms such as MT-Bench and Chatbot Arena (see Section 2.4.3). In their experimental evaluation, the team found that a GPT-4-based judge achieved an 80% agreement rate with human evaluators, which is comparable to the consensus level typically observed between human judges.

As illustrated in Figure 2.22, the study presents an example of multi-turn evaluations between two user-assistant models, showcasing the mechanics of this new paradigm in practice [225].

Question: If the FED buys bonds in the secondary market
(A) the money supply will increase. (B) the money supply will decrease. (C) the money supply will not be affected. (D) the discount rate would be affected. Output the answer only.

Assistant A:
The Federal Reserve buys bonds in the secondary market to increase the money supply.

Assistant B:
(A) the money supply will increase.

Follow-up Question: How does it affect my daily life? Give 3 examples.

Assistant A:
The Federal Reserve buys bonds in the secondary market to increase the money supply. This affects your daily life by:
1. Increasing the money supply.
2. Increasing the money supply and lowering the discount rate.
3. Increasing the money supply and lowering the discount rate and increasing the money supply.

Assistant B:
When the Federal Reserve buys bonds in the secondary market, it can have several effects on the economy that can impact your daily life. Here are three examples:
1. Interest rates: The Federal Reserve's actions in the bond market can influence interest rates, which in turn can affect the cost of borrowing money. For example, if the Federal Reserve buys bonds and drives up demand for them, interest rates on things like mortgages, car loans, and credit cards may go up, making it more expensive to borrow money.
2. Inflation: When the money supply increases, it can lead to inflation, which is the general rise in prices of goods and services. This can impact your daily life by making it more expensive to buy things like groceries, gas, and other necessities.
3. Employment: The Federal Reserve's actions in the bond market can also have an impact on employment. For example, if the Federal Reserve's actions lead to economic growth and increased demand for goods and services, it can create more job opportunities and boost employment.

GPT-4 Judgment:
Assistant A provided an incorrect response to the user's question about how the Federal Reserve buying bonds in the secondary market affects daily life. The answer given is repetitive and lacks clear examples of how the action impacts daily life.
On the other hand, Assistant B provided a relevant and accurate response to the user's question about the Federal Reserve buying bonds. The answer includes three clear examples of how the action impacts daily life, such as interest rates, inflation, and employment.

Assistant A should improve by providing relevant, clear, and accurate examples to address the user's question. Assistant B's response is more helpful, relevant, and detailed.

Figure 2.22: Multi-turn dialogues between a user and two AI assistants—*LLaMA-13B* (Assistant A) and *Vicuna-13B* (Assistant B). Source: [225]

The research also highlighted significant limitations of traditional benchmarking approaches. For instance, while models like *LLaMA-13B* and *Vicuna-13B* demonstrated competitive performance on conventional benchmarks, **they struggled considerably when confronted with open-ended tasks**, receiving markedly fewer positive evaluations from human judges.

The study uncovered several significant biases intrinsic to LLM-based judging, each influencing the evaluation process in distinct ways. One such bias is *position bias*, where the first response presented is often favored, irrespective of its actual content quality. Additionally, *verbosity bias* reveals a preference for longer, more detailed responses, even when shorter, clearer answers might offer superior quality. Lastly, *self-enhancement bias* emerges when LLM judges tend to favor responses generated either by themselves or by smaller models, potentially leading to conflicts of interest in the evaluation process.

Additionally, the researchers noted persistent **challenges** in the evaluation of mathematical and reasoning-based questions, where the LLMs demonstrated limited capabilities in grading due to their

inability to compute or verify the correct answers. This limitation underscores the current inefficacy of LLMs in domains requiring precise calculations or logical deductions.

To mitigate these challenges, *Zeng et al.* proposed several solutions, including *position swapping*, employing few-shot judging techniques to enhance consistency, utilizing Chain-of-Thought prompting strategies, and incorporating reference-guided judging. They also suggested fine-tuning specialized LLMs to act as more reliable judges. Based on their findings, they produced a public leaderboard showcasing the comparative performance of various models, which can be accessed online.²

Following this foundational work, several alternative approaches have emerged. These include prompt-based methods such as *G-Eval* [2] and *SelfCheckGPT* [118], as well as models fine-tuned specifically for evaluation, such as *CriticGPT* [119]. These innovations continue to build on the “LLM-as-a-Judge” framework, pushing the boundaries of LLM evaluation techniques and striving for increased fairness, consistency, and accuracy in model assessment.

²<https://huggingface.co/spaces/lmarena-ai/chatbot-arena-leaderboard>

2.4.5 RAG Evaluation

In previous sections, we have discussed the evaluation of the final component in the RAG pipeline: the Large Language Model. However, the question arises—*how should we evaluate the entire Retrieval-Augmented Generation system?*

First, it is important to review the primary downstream tasks associated with RAG (49). These tasks include *Question Answering* (encompassing single/multi-hop, multiple choice, domain-specific, and graph-based variants), *Dialog*, *Information Extraction*, *Reasoning*, and other classical NLP tasks as outlined in subsections 2.4.3 and 2.4.1. Each of these categories has its own dedicated datasets and evaluation methodologies. Due to the breadth of the subject, Table 2.8, as compiled by *Gao et al.* [49], provides an overview of the primary contributions in terms of datasets and evaluation methods for each downstream task.

A particular emphasis must be placed on **Quality Scores**, which assess three key aspects: *Context Relevance*, *Answer Faithfulness*, and *Answer Relevance*. These can be summarized as follows:

- **Context Relevance** evaluates the accuracy and specificity of the retrieved context, ensuring that only relevant information is used, thereby minimizing computational overhead due to irrelevant content.
- **Answer Faithfulness** ensures that the generated answers remain consistent with the retrieved context, preventing contradictions and preserving coherence.
- **Answer Relevance** focuses on the alignment of the generated answers with the posed questions, ensuring that the answers effectively address the core inquiry.

Low quality scores in any of these aspects can cause a well-functioning LLM to produce suboptimal results, underscoring the importance of careful evaluation.

Key Abilities for RAG Systems

In addition to evaluating quality scores, RAG systems must also exhibit a set of core abilities that are essential for ensuring adaptability and robust performance. These abilities include:

- **Noise Robustness:** The capacity to manage documents that may be related to the query but offer little useful information, maintaining focus on relevant content.
- **Negative Rejection:** The discernment to avoid producing a response when the retrieved documents do not provide adequate knowledge to answer the query.
- **Information Integration:** The ability to synthesize information from multiple sources to construct a well-rounded and accurate response to more complex inquiries.
- **Counterfactual Robustness:** The skill to identify and disregard inaccuracies within retrieved documents, especially when dealing with misleading or incorrect data.

Both *Context Relevance* and *Noise Robustness* are crucial for assessing the quality of retrieval, while *Answer Faithfulness*, *Answer Relevance*, *Negative Rejection*, *Information Integration*, and *Counterfactual Robustness* are pivotal for evaluating the quality of the generation process.

The metrics used to evaluate these aspects are outlined in Table 2.6.

Benchmarks and Tools

Several prominent benchmarks, including *RGB* [19], *RECALL* [109], and *CRUD* [112], are specifically designed to assess the essential capabilities of RAG models. In addition, advanced automated tools such as *RAGAS* [45], *ARES* [146], and *TruLens* leverage LLMs to evaluate quality scores.³ Together, these benchmarks and tools provide a comprehensive framework for systematically evaluating the performance of RAG systems, as summarized in Table 2.7.

³<https://github.com/truera/trulens>

Table 2.6: Summary of metrics applicable for evaluation aspects of RAG. (Source: [49])

	Context Relevance	Faithfulness	Answer Relevance	Noise Robustness	Negative Rejection	Information Integration	Counterfactual Robustness
Accuracy	✓	✓	✓	✓	✓	✓	✓
EM					✓		
Recall	✓						
Precision	✓			✓			
R-Rate							✓
Cosine Similarity			✓				
Hit Rate	✓						
MRR	✓						
NDCG	✓						
BLEU	✓	✓	✓				
ROUGE/ROUGE-L	✓	✓	✓				

Table 2.7: Summary of evaluation frameworks (Source: [49])

Evaluation Framework	Evaluation Targets	Evaluation Aspects	Quantitative Metrics
RGB	Retrieval Quality Generation Quality	Noise Robustness Negative Rejection Information Integration Counterfactual Robustness	Accuracy EM Accuracy Accuracy
RECALL	Generation Quality	Counterfactual Robustness	R-Rate (Reappearance Rate)
RAGAS	Retrieval Quality Generation Quality	Context Relevance Faithfulness Answer Relevance	* * Cosine Similarity
ARES	Retrieval Quality Generation Quality	Context Relevance Faithfulness Answer Relevance	Accuracy Accuracy Accuracy
TruLens	Retrieval Quality Generation Quality	Context Relevance Faithfulness Answer Relevance	*
CRUD	Retrieval Quality Generation Quality	Creative Generation Knowledge-intensive QA Error Correction Summarization	BLEU ROUGE-L BertScore RAGQuestEval

Table 2.8: Downstream tasks and datasets of RAG. (Source: [49])

Task	Sub Task	Dataset	Method
QA	Single-hop	Natural Qustion(NQ) [84]	[22, 78, 5, 10, 210, 195, 65, 143, 180, 47] [144, 73, 34, 106, 92, 108, 63, 3, 163, 200] [157, 181, 24]
		TriviaQA(TQA) [75]	[22, 210, 195, 212, 143, 180] [47, 144, 106, 92, 108] [63, 163, 200, 157, 181, 4]
		SQuAD [141]	[22, 79, 180, 206, 108, 52, 24]
		Web Questions(WebQ) [9]	[22, 67, 210, 212, 92, 3]
		PopQA [117]	[204, 114, 4]
		MS MARCO [126]	[10, 73, 92]
Multi-hop	HotpotQA	[207]	[78, 5, 202, 111, 195, 213, 79, 173] [154, 47, 192, 144, 206, 106, 163, 200, 114]
		2WikiMultiHopQA [62]	[72, 173, 154, 47, 192, 209]
		MuSiQue [174]	[202, 173, 154, 192]
Long-form QA	ELI5 [46]	[46]	[202, 195, 106, 17, 63]
		NarrativeQA(NQA) [82]	[151, 88, 180, 201]
		ASQA [162]	[72, 80]
		QMSum(QM) [227]	[88, 201]
Domain QA	Qasper [35]	[35]	[151, 201]
		COVID-QA [123]	[152, 159]
		CMB [188], MMCU_Medical [218]	[71]
Multi-Choice QA	QuALITY [134]	[134]	[151, 201]
		ARC [31]	[204, 4]
		CommonsenseQA [166]	[191, 194]
Graph QA	GraphQA [58]	[58]	[58]
Dialog	Dialog Generation	Wizard of Wikipedia (WoW) [40]	[195, 65, 212, 106]
		KBP [183]	[184, 185]
	Task-oriented Dialog	DuleMon [203]	[185]
		CamRest [198]	[156, 155]
IE	Event Argument Extraction	Amazon(Toys,Sport,Beauty) [57]	[73, 142]
		WikiEvent [97]	[41, 65, 212, 106]
	Relation Extraction	RAMS [43]	[115, 41]
		T-REx [44], ZsRE [90]	[202, 106]
Reasoning	Commonsense Reasoning	HellaSwag [215]	[24, 194]
		CoT Reasoning [81]	[106]
		CSQA [147]	[221]
Others	Language Understanding	MMU [60]	[213, 65, 133, 106, 63, 157, 114]
		WikiText-103 [121]	[85, 143, 200, 12]
	Fact Checking/Verification	StrategyQA [51]	[202, 72, 221, 154, 209, 191]
		FEVER [170]	[210, 195, 65, 212, 106, 92]
	Text Generation	PubHealth [83]	[4, 204]
		Biography [87]	[204]
	Text Summarization	WikiASP [56]	[72]
		XSum [125]	[25]
	Text Classification	VioLens [148]	[98]
		TREC [99]	[54]
	Sentiment	SST-2 [160]	[54, 24, 187]
		CodeSearchNet [64]	[101]
	Robustness Evaluation	NoMIRACL [169]	[169]
		GSM8K [32]	[120]
	Math	JRC-Acquis [161]	[25]
	Machine Translation		

2.4.6 Challenges

Evaluating LLMs presents numerous challenges, particularly as their capabilities surpass traditional natural language tasks. While automatic evaluations provide crucial insights, several key challenges arise in human evaluation that need to be addressed for comprehensive and reliable assessments [16].

One of the most significant challenges is the design of suitable benchmarks to measure *Artificial General Intelligence* (AGI). Current benchmarks are often inadequate for evaluating the full range of AGI’s potential since they typically focus on specific tasks. A broader, interdisciplinary understanding, incorporating fields such as psychology, education, and social sciences, is essential to construct benchmarks that effectively assess AGI capabilities. A central question remains: *should human values serve as the foundation for AGI benchmarks, or should alternative perspectives be adopted?* This presents unresolved research challenges regarding the development of AGI-specific evaluation protocols.

Another important aspect is the need for *complete behavioral evaluation*. Standard benchmarks typically assess LLMs on static tasks, but AGI models should also be evaluated in dynamic and open environments that simulate real-world scenarios. For instance, by treating LLMs as central controllers, evaluations could be performed on robots manipulated by these models to test their behavior in practical situations. Additionally, the evaluation of multi-modal dimensions, as AGI models will likely interact with various data types, is essential for a comprehensive assessment.

Robustness evaluation also remains a critical issue. LLMs need to demonstrate consistent performance across a wide range of inputs, including different linguistic expressions. However, current models like ChatGPT display variability in outputs when presented with prompts that differ only in grammar or phrasing, exposing the models’ lack of robustness. This underscores the need for a more robust evaluation framework, including a broader set of input types and dynamic robustness tests that evolve alongside the models’ progress. Furthermore, evolving definitions of robustness, particularly concerning ethics and bias, highlight the need for continuous updates to the evaluation system.

The reliance on *static benchmarks* poses another challenge. Current evaluation protocols are often based on static, publicly available datasets, which do not account for the rapidly evolving capabilities of LLMs. Static benchmarks may eventually become outdated or be memorized by LLMs, leading to biased or inaccurate evaluations. Therefore, developing *dynamic and evolving evaluation systems* is key to ensuring a fair and accurate assessment of LLMs as they continue to advance.

Another critical challenge is ensuring *principled and trustworthy evaluation*. It is essential to scrutinize the integrity of the evaluation systems themselves. Trustworthy evaluation systems must guarantee that models are tested on out-of-distribution examples and avoid contamination from training data. This requires further research into measurement theory and probability, as well as a deeper understanding of how to evaluate the reliability of these systems.

Additionally, there is a need for a *unified evaluation system* that supports the wide variety of tasks performed by LLMs. These evaluation systems must be capable of handling tasks such as value alignment, safety verification, interdisciplinary research, and fine-tuning. For example, systems like *PandaLM* [190] offer open-source evaluation models to assist in the fine-tuning of LLMs, but broader systems are required to support the increasing diversity of LLM applications.

Finally, evaluation should not be the ultimate goal but rather the starting point for *LLM enhancement*. A robust evaluation system should provide not only benchmark results but also actionable insights and recommendations for future research. For instance, *PromptBench* [230] offers detailed analyses of robustness results, along with recommendations for prompt engineering, thereby guiding model improvements. Similarly, other studies have explored how performance evaluations can lead to enhancements in specific tasks. In this way, evaluation serves as a catalyst for refining and advancing LLMs.

2.5 Telecommunication concepts

This section provides a brief overview of key concepts within the telecommunications domain, particularly those related to the capabilities of 5G networks. These concepts are critical for understanding the functionality of Nokia's "Network as Code" SDK, as they offer the technological context within which the system operates.

2.5.1 5G Networks

5G, the fifth generation of wireless communication technology, marks a significant evolution beyond its predecessor, 4G LTE, offering not only enhanced speed but also transformative improvements in network capacity, latency, and reliability. Unlike previous generations that primarily focused on increasing data rates, 5G is designed to accommodate a wide array of services and use cases, each with distinct performance requirements. Central to this advancement are key innovations such as *millimeter wave (mmWave)* technology, massive *MIMO (Multiple Input Multiple Output)*, and *beamforming*, which collectively enable 5G to achieve ultra-fast data transfer speeds, support a significantly higher density of connected devices, and minimize end-to-end latency.

These technological breakthroughs are not merely incremental improvements; they are foundational for enabling next-generation applications that demand high performance and responsiveness. For instance, 5G's capabilities are essential for the proliferation of the *Internet of Things (IoT)*, where millions of devices—ranging from sensors to smart appliances—must communicate efficiently in real time. Similarly, industries such as autonomous vehicles, *augmented reality (AR)*, and industrial automation rely on 5G's low-latency, high-reliability features to operate safely and effectively. By providing a flexible, scalable, and efficient architecture, 5G establishes the necessary infrastructure to drive digital innovation across a diverse range of sectors, serving as the backbone for future technological advancements.

2.5.2 Programmable Networks

At the core of 5G's flexibility are programmable networks, which allow network operators to adapt and optimize their infrastructure in real-time. Using technologies such as Software-Defined Networking (SDN) and Network Function Virtualization (NFV), 5G networks decouple the control and data planes, providing centralized control over the entire network. This decoupling enables dynamic reconfiguration of the network to meet specific service requirements, such as altering traffic flows or provisioning additional resources. In a programmable network, operators can swiftly deploy new services, modify existing ones, and implement security or traffic policies without the need for hardware changes. This flexibility is critical in 5G, where the network must support a variety of applications, each with different demands for bandwidth, latency, and reliability.

2.5.3 Quality of Service on Demand

5G networks introduce advanced mechanisms to deliver Quality of Service (QoS) on demand, ensuring that network resources are allocated according to the specific needs of different services and applications. QoS refers to the ability of the network to guarantee particular performance metrics, such as bandwidth, latency, jitter, and packet loss, which are crucial for delivering consistent and reliable service. In 5G, the diverse use cases, such as augmented reality (AR), real-time video streaming, remote healthcare, and smart grids, require different QoS levels. Through the use of network programmability and slicing, 5G can dynamically adjust and guarantee QoS parameters for each service, enabling seamless user experiences even in congested network environments. This dynamic QoS management is essential for maintaining high service levels across a variety of applications.

2.5.4 Network Slicing

One of the most innovative features of 5G is network slicing, which allows the creation of multiple virtual networks, or slices, on a single physical infrastructure. Each slice is tailored to serve specific business or application needs, providing a customized set of network resources and Quality of Service (QoS) guarantees. For example, a slice for IoT devices may prioritize low-power consumption and connection density, while a slice for autonomous vehicles would focus on ultra-low latency and high reliability. Network slicing enables the isolation of different services, preventing performance degradation due to interference from other traffic types. This capability ensures that 5G can simultaneously

support a wide range of industries, from healthcare and manufacturing to entertainment and transportation, each with its own unique performance requirements. The ability to deliver such customized, on-demand services makes 5G a key enabler of digital transformation across multiple sectors.

3 Methodology

This section outlines the methodologies used to develop the evaluation tool for Large Language Models. First, a brief overview of the core elements underlying the tool will be presented, specifically the *Network as Code* (NaC) SDK and *Copilot*—a flexible Retrieval-Augmented Generation pipeline designed to work with Nokia’s internal documentation as well as other compatible datasets.

Following this, the system architecture will be examined, with a particular emphasis on the tool’s interface and operational flow. This section will demonstrate how users initiate the evaluation process from an initial set of questions, ultimately enabling a comprehensive assessment of the LLM’s performance through structured interaction with the tool.

3.1 Setup

In order to evaluate the LLMs, it has been necessary to build a pipeline that allows users to generate a dataset of questions and answers based on the context (in this case, Network as Code), create derivative datasets from the original, and, finally, to employ a tool for computing metrics and displaying the results. The high-level pipeline, encompassing the collection of the golden dataset through to the evaluation of the children datasets, is illustrated in Figure 3.1.

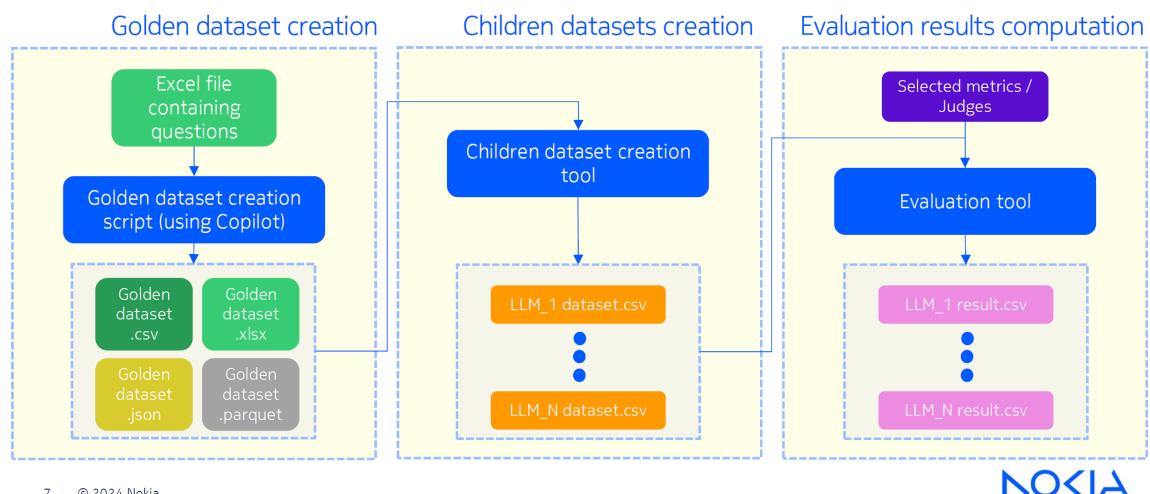


Figure 3.1: Full evaluation pipeline (from data gathering to children dataset evaluation)

3.1.1 Data source: Network as Code (NaC) SDK documentation

The Network as Code (NaC) SDK developer documentation is the primary data source upon which this evaluation tool has been developed. NaC introduces a new approach to network management, transforming traditional static networks into flexible environments capable of adjusting to application requirements. By abstracting complex network operations, the NaC SDK makes network programming accessible to developers without specialized telecom expertise, allowing interaction with 5G capabilities through high-level APIs.

The NaC SDK provides a set of functionalities designed to enhance control, flexibility, and efficiency in network interactions:

- **Simplified Network Access:** Through code abstractions, the NaC SDK allows developers to manage network resources, request specific capabilities, and make adjustments without needing deep telecom knowledge.

- **Data Flow Prioritization:** Applications can dynamically prioritize data flows (download or upload), making it easier to manage network resources for varying application needs. For latency-sensitive applications such as *Internet of Things (IoT)* or *Virtual Reality (VR)*, the SDK also supports Quality-of-Service on Demand (**QoD**) to ensure stable performance.
- **Resource Optimization:** The SDK enables devices to adjust bandwidth dynamically, reducing it during idle periods or scaling it up in response to high traffic, which can save both bandwidth and power.
- **Location-Based Features:** Developers can access geolocation data (latitude, longitude, and radius) for devices, supporting use cases such as fraud prevention or smart mobility, where real-time location information is essential.
- **Device Connectivity Monitoring:** The SDK provides real-time monitoring of device connectivity states (roaming, attachment, or detachment) and allows status updates to be posted to specified URLs, which helps applications adjust to connectivity changes.
- **Specialized Network Slices:** The SDK allows creation of dedicated network slices for customized requirements, which is critical for applications in VR, *Extended Reality (XR)*, and other high-demand fields, ensuring quality and security for specialized services.

The NaC SDK, available in both TypeScript and Python programming languages, provides a wide range of network functionalities that simplify the integration of advanced 5G capabilities into applications. With these features, developers can effectively manage network resources and tailor network interactions to meet specific application requirements.

3.1.2 Data gathering with Copilot (Nokia's internal assistant)

To create a dataset for querying and evaluating various Large Language Models, an internal tool developed by Nokia, called *Copilot*, was utilized. It is crucial to distinguish this tool from *Microsoft Copilot*¹ as Nokia's Copilot is a distinct solution designed specifically for internal use. This tool operates as a RAG pipeline, enabling interaction with an LLM to retrieve precise information from an indexed corpus of documentation stored within a vector database.

The pipeline, illustrated in Figure 3.2, comprises the following components:

- a query embedding model (*bge-m3*) that represents text as vectors;²
- a vector database (*Milvus*) containing all documentation information as vectorized chunks, i.e., smaller segments of the original documents;³
- a reranker model (*bge-reranker-v2-m3*);⁴
- a dedicated system prompt template that integrates the given chunks;
- an open-source LLM hosted on proprietary servers (*LLaMA 3.1 70B Instruct*).⁵

¹<https://copilot.microsoft.com/>

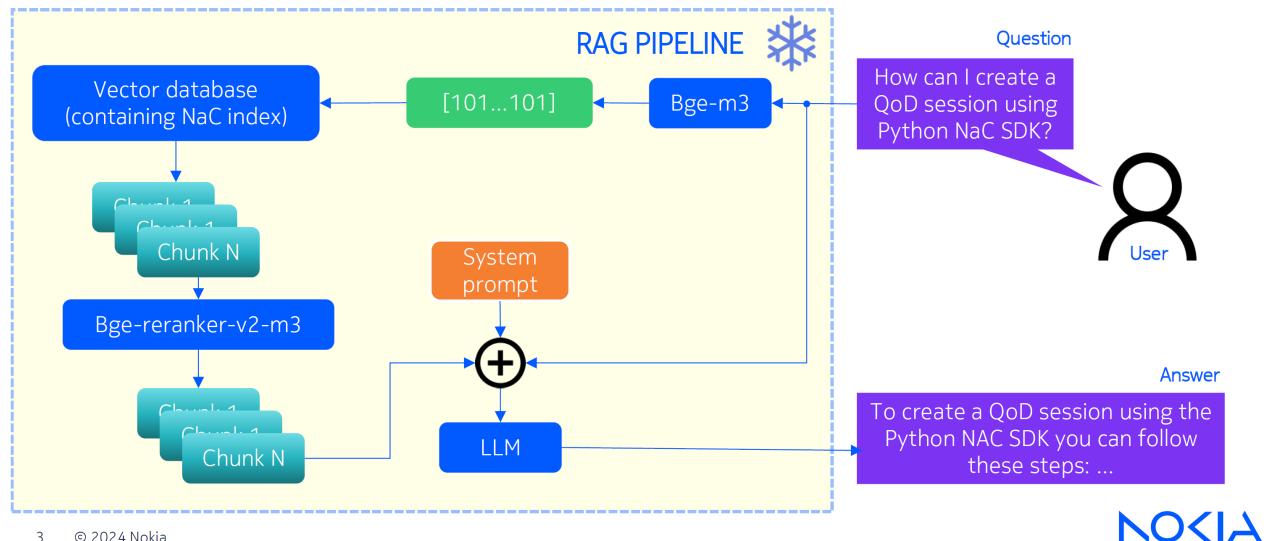
²<https://huggingface.co/BAAI/bge-reranker-v2-m3>

³<https://github.com/milvus-io/milvus>

⁴<https://huggingface.co/BAAI/bge-reranker-v2-m3>

⁵<https://huggingface.co/meta-llama/Llama-3.1-70B>

We want to evaluate data that are obtained from “frozen” RAG setup



3 © 2024 Nokia

Figure 3.2: Nokia’s Copilot frozen RAG pipeline used in the data gathering process

Table 3.1: RAG Pipeline Hyperparameters with choice justification

Hyperparameter	Value	Reason
Temperature	0.1	Limits LLM creativity and enforces reproducibility, aiming for predictable answers without making them entirely robotic.
Database choice	Milvus	Supports both sparse and dense vector representation: sparse for lexical (keyword, term) results, and dense for semantic (sentence, meaning) results.
Embedding model	BAAI/bge-m3	High-quality multilingual embedding model that can produce both sparse and dense embeddings, with long context handling for large text chunks.
Reranker model	BAAI/bge-reranker-v2-m3	Corresponding reranker model to sort semantic results by relevance with the user query.
Minimum semantic relevance	0.01	Experimental cut-off point for low-relevance semantic results. Document chunks with lower scores are not sent to the LLM as knowledge.
Large Language Model	meta-llama/Meta-Llama-3.1-70B-Instruct	Provides the best available cost/quality tradeoff.
LLM output max tokens	2048	A 2048-token limit (approx. 8000 characters) typically fits retrieved Q and A content in RAG; 4096 or 8192 tokens may be better for long-form content generation.
Max chunk size [chars]	6000	Targets chunking one document section (header + content) or table per chunk. Small sections/tables can be concatenated.
Max knowledge size [chars]	60000	Best tradeoff between inference time, performance, and quality, leaving room for conversation history within a 32k or 64k context limit.
Chunks overlap	False	No need for redundant text in chunks, as chunks are split by document sections (header + content or entire table).
Query reformulation (surrogate query)	True	Rewriting the user query enhances search effectiveness by targeting semantic, lexical, or hybrid queries optimized for the database query engine.

It is important to note that the pipeline is considered “*frozen*”, meaning no hyperparameters—such as *chunk size*, *embedding dimensions*, or *temperature* values—have been modified to enhance performance. The specific hyperparameters, their values, and the justification for each choice are reported in Table 3.1. This evaluation tool, by design, does not alter the entire RAG pipeline, as doing so would introduce a high level of complexity. Instead, given its *proof-of-concept* nature, the tool serves exclusively as a platform for evaluation at the final stage of the process: the large language model.

In the initial phase of data gathering, all the Markdown-formatted documents from the SDK’s online documentation were indexed.⁶ Following this, a script was developed to create the “**golden dataset**”. The term *golden dataset* refers to a dataset containing the *User Query* (or question), the *System Prompt* (a template incorporating LLM instructions along with the chunks retrieved from the vector database), the *LLM’s response*, and additional metadata such as the question category and whether the code provided should be executed. An example of a chunk is reported in Listing 3.1.

Although closed-source models like *Claude 3.5 Sonnet*⁷ and *GPT-4o*⁸ sometimes outperform open-source models, the LLaMA 3.1 70B Instruct model was chosen as the reference model for generating the golden dataset. This decision was influenced by both technical setup considerations and budget constraints. Specifically, the platform on which these closed-source models were hosted (*Azure AI*)⁹ encountered errors due to context window limitations. Consequently, the most capable model available with a flexible context window was Meta’s open-source LLaMA model.

In this experiment, we operate under the **assumption** that **LLaMA 3.1 70B Instruct has produced the most accurate results**, and, therefore, **the entries in this dataset are regarded as the most correct and reliable for evaluation purposes**.

In total, **62 prompts** were collected, divided into **8 categories** (as detailed in Table 3.2 and Figure 3.3). Notably, **22** of these prompts were classified as **containing executable code to be evaluated**. It should be noted, however, that these are not the only prompts involving executable code. For example, in the “*Specialized Networks*” category, the decision was made to allow the evaluation tool to execute only a subset of the generated scripts. This choice was primarily due to the low-level slicing operations these scripts perform on the network, which demand considerable execution time.

Therefore, only the core functions for specialized network tasks were executed, under the assumption that if these core functions perform correctly, the LLM should also be capable of handling similar cases with minor variations. Such cases might include specifying a slice ID or adjusting throughput parameters.

Listing 3.1: Chunk containing Location Retrieval Code Example

```
---
element_id: e5d7af26-405e-4fd8-bc7e-fd793a321899
metadata:
  product_id: none
  filetype: text/plain
  filename: location-retrieval.mdx
  last_modified: 2024-09-25T07:51:49
---

--- tags: location retrieval, location retrieval SDK, getting device geographic coordinates desc:
  Learn how to retrieve the geographic coordinates or location information of a device ---
# # Location retrieval
This feature allows you to get the geographic coordinates of a device and exhibit them on the screen.
  You will also need to specify a certain amount of time or maximum time limit for the location
  information you receive. This way, your device's location will always be up to date!
> **NOTE:** Querying a device's location might have serious privacy implications. > So, it should be
  done carefully, and strict authorization may be needed.
## Retrieving and displaying a device's location
```

⁶<https://developer.networkascode.nokia.io/docs>

⁷<https://www.anthropic.com/news/clause-3-5-sonnet>

⁸<https://openai.com/index/hello-gpt-4o/>

⁹<https://azure.microsoft.com/en-us/solutions/ai>

Retrieving the location of a device can be fairly simple. Assuming a device was already created in [this previous step](/docs/location/location-verification), a ‘location’ object, as the ones below, can be instantiated from the fields for longitude and latitude.

You can use the following code snippets to exhibit a device’s geographic coordinates and address on the screen:

```
# <Tabs>
# <Tab label="Python">
# ````python
# import network_as_code as nac
# from network_as_code.models.device import DeviceIpv4Addr
# We initialize the client object with your application key client = nac.NetworkAsCodeClient( token="<your-application-key-here>", )
my_device = client.devices.get( "device@testcsp.net", ipv4_address=DeviceIpv4Addr( public_address ="233.252.0.2", private_address="192.0.2.25", public_port=80 ), ipv6_address="2001:db8:1234:5678:9abc:def0:fedc:ba98", # The phone number does not accept spaces or parentheses phone_number ="+36721601234567" )
# Specify the maximum amount of time accepted # to get location information, it's a mandatory parameter. # If the amount in seconds is not given, the default will be 60 seconds. location = my_device.location(max_age=3600)
# The location object contains fields for longitude, latitude and also elevation longitude = location.longitude
latitude = location.latitude
print(longitude) print(latitude) print(location.civic_address) ```` </Tab> <Tab label="TypeScript"> ````typescript
import { NetworkAsCodeClient } from 'network-as-code';
// We initialize the client object with your application key const client = new NetworkAsCodeClient("<your-application-key-here>");
// Create a device object for the mobile device we want to use const myDevice = client.devices.get({
  networkAccessIdentifier: "device@testcsp.net", ipv4Address: { publicAddress: "233.252.0.2",
  privateAddress: "192.0.2.25", publicPort: 80, }, ipv6Address: "2041:0000:140F::875B:131B", // The phone number does not accept spaces or parentheses phoneNumber: "+36721601234567" });
// Specify the maximum amount of time accepted // to get location information, it's a mandatory parameter. // The value (integer in seconds) can be passed directly. // If the amount in seconds is not given, the default will be 60 seconds. const location = await myDevice.getLocation(3600);
// The location object contains fields for longitude, latitude and also elevation console.log(location.latitude); console.log(location.longitude); console.log(location.civicAddress); ```` </Tab> </Tabs>
>
**What does this code do?**
It prints your device's current latitude, longitude coordinates.
> **NOTE:** The civic address is an optional field in the query response. > If it is set in the API, then it will return the device's current civic address. > In case it is not set, it may also return 'None' on 'undefined' values.
This code also creates a Network as Code client using an application key accessible from your [**Dashboard**](/dashboard) and identifies a mobile network device in multiple ways (IP addresses, port, etc). Learn more about the [client](/docs/general-concepts/network-as-code-client) and [device](/docs/general-concepts/network-as-code-client#creating-a-device-object) objects.
---
```

During the initial attempts to construct the golden dataset, it became apparent that, despite the System Prompt’s level of detail, certain instructions were frequently overlooked by the LLM. For instance, the LLM would often neglect directives to produce only an end-to-end code snippet or to maintain a consistent device configuration throughout code generation. Consequently, within the dataset generation script, “additional instructions” were incorporated based on the specific prompt type to reinforce the intended behavior. This addition aimed to address the issue wherein certain instructions were diluted when embedding chunks into the System Prompt.

Some examples of these additional instructions include:

- Please provide a descriptive answer without including any code snippet.
- Display the result using a Markdown table. Please provide a descriptive answer without including any code snippet.
- Please provide just only end-to-end one code snippet. Print only strings concerning the performed operation like as you are debugging (e.g., "client created"). Do not print variables and do not guess or make up attributes and methods.

```

<instruction>
// IMPORTANT ADDITIONAL INSTRUCTIONS:
// - When asked about operations in Slice management, please always use async code (use asyncio
//   library if Python) and remember to respect the associated lifecycle. Use "await" only when
//   waiting for a specific state. Do not use "await" to call other functions.
// - If you created either a QoD, a Slice, or a Subscription, always delete them with the .
//   delete() method so that your code can be self-contained and independent from previous or
//   future snippets.
**Slice Lifecycle:** A slice (or specialized network) lifecycle consists of creation, activation
, deactivation, and removal, with possible states including 'PENDING', 'AVAILABLE', '
OPERATING', 'DELETED', or 'FAILED', requiring careful management to ensure proper transition
between states.
// - When providing code, always include all the necessary imports and ensure the code is
executable standalone, independent of previous actions, even though a history is provided.
Always follow the suggestions provided as code comments.
// - Plan before generating code, explain your Chain of Thoughts (reasoning), and then provide
the code in just one snippet. Do not split the code while thinking.
// - Do not forget to properly include DeviceIpv4Addr
// - When generating code, if not previously defined, always use the following configuration
when getting a device:
"device@testcsp.net",
ipv4_address=DeviceIpv4Addr(
    public_address="233.252.0.2",
    private_address="192.0.2.25",
    public_port=80
),
ipv6_address="2001:db8:1234:5678:9abc:def0:fedc:ba98",
phone_number="+36721601234567"
// Always use phone number with "+" before the actual number (it's a requirement!)
// - When inserting the token, always provide it in this way: token=<your-application-key-here
>"
</instruction>

```

These targeted instructions were designed to guide the LLM more effectively, ensuring that the responses align with the desired format and specifications of the golden dataset. To facilitate interoperability—especially among less technically oriented business units—the prompt file used to generate the golden dataset was stored in *Microsoft Excel* (.xlsx) format. However, after generation, the golden dataset can be exported in various formats, including .xlsx, .parquet, .json, and .csv, as outlined in the preceding methodology. It is worth noting that the .xlsx format occasionally exhibits cell content "clipping," likely due to field length constraints intrinsic to the format. To address this, alternative file formats are preferred where feasible to preserve data integrity and accessibility across diverse use cases.

The workflow of the golden dataset creation is represented in Figure 3.4, illustrating each stage of the process, from data gathering to the final dataset generation.

Table 3.2: Prompt Categories and Descriptions

Category Name	Prompts Count	Description
Device Status	4	Monitor when a device is attached to or detached from a mobile data network.
General Concepts	12	Fundamental concepts to understand SDK capabilities.
Identity and Security	4	SIM Swap API detection-related questions.
Initialization	3	Handle client and device initialization.
Localization	5	Localization verification and retrieval API usage.
Network Insights	5	Manage congestion notifications and subscriptions.
QoS	8	Quality-of-Service profiles, on-demand session, and lifecycle management.
Specialized Networks	21	Slices lifecycle management and device/application attachment/detachment to slices.

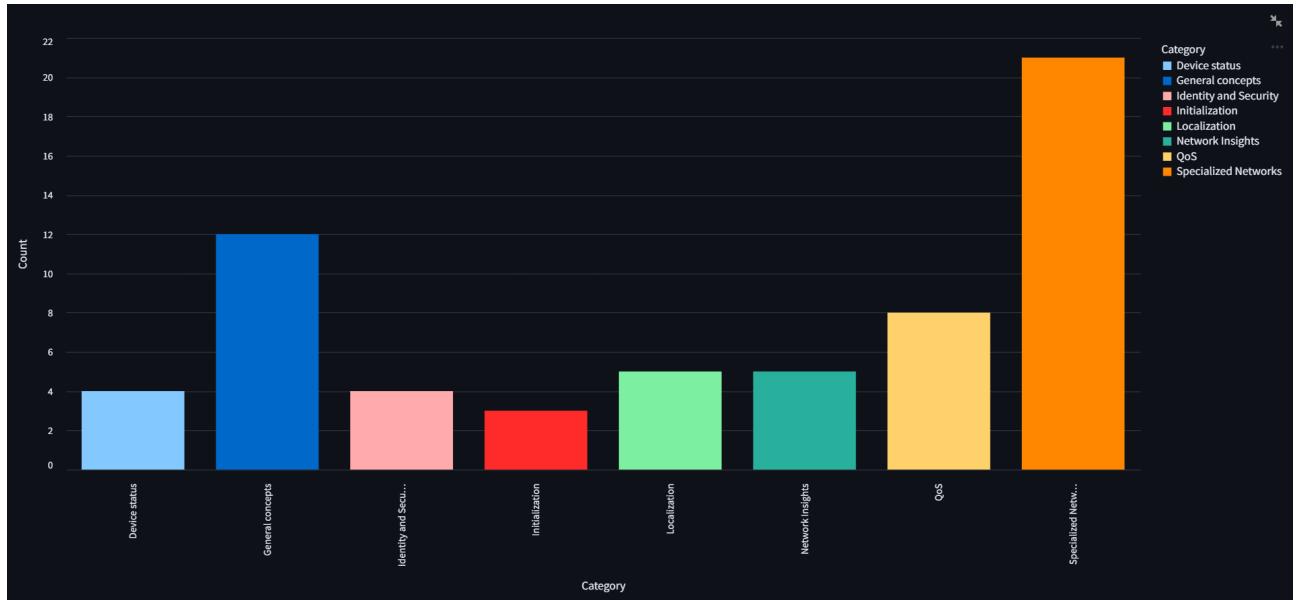
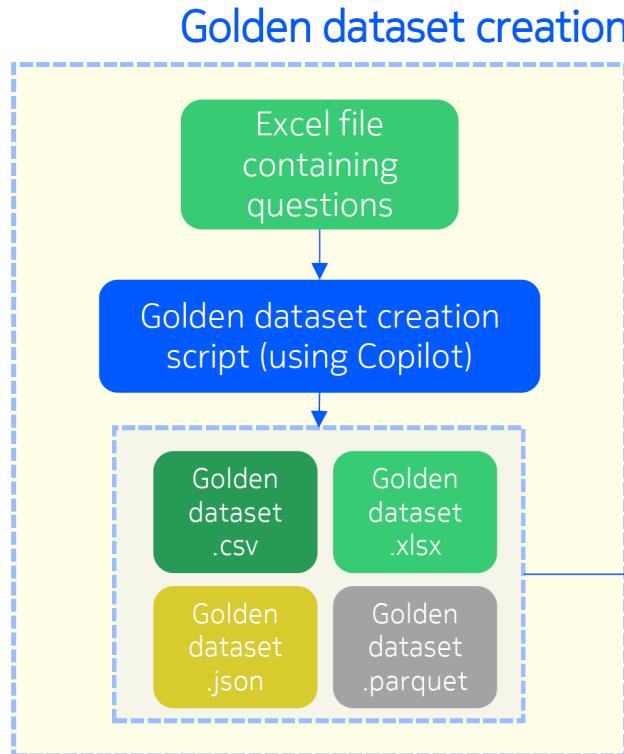


Figure 3.3: Golden dataset prompts categories

3.1.3 Additional Tools Utilized

The development of the LLM evaluation application required third-party software to facilitate task abstraction and manage concurrent operations effectively. Table 3.3 provides an overview of the primary tools employed for each task.



9 © 2024 Nokia

Figure 3.4: Golden dataset creation workflow

Table 3.3: Key Tools Used for Each Task

Task	Tool
Reference LLM inference for golden dataset generation	vLLM ¹ (hosted on a local server with a dedicated NVIDIA H100 GPU) ²
Inference of open-source LLMs	Ollama ³ (hosted on a local server with a dedicated NVIDIA A40 GPU) ⁴
Inference of closed-source LLMs (e.g., GPT-4o, GPT-4o mini, Claude 3.5 Sonnet)	Azure AI ⁵
Frontend development for the evaluation application	Streamlit ⁶

¹ <https://github.com/vllm-project/vllm>
² <https://www.nvidia.com/en-us/data-center/h100/>
³ <https://github.com/ollama/ollama>
⁴ <https://www.nvidia.com/en-us/data-center/a40/>
⁵ <https://azure.microsoft.com/en-us/solutions/ai>
⁶ <https://github.com/streamlit/streamlit>

3.2 Evaluation Tool

The evaluation tool is a multi-page application developed in *Python*, utilizing the *Streamlit* framework for its frontend interface. Each page is purposefully designed to execute a specific step, thereby guiding users toward achieving the final evaluation results.

In the *NaC Dataset Explorer* page, users can access the original dataset of questions in *.xlsx* format (excluding answers), the golden dataset in the formats outlined in Section 3.1.2, and the various derived datasets, hereafter referred to as "*children datasets*". This page not only facilitates

the visualization of existing datasets but also allows users to generate new datasets derived from the golden dataset. These new datasets will represent responses generated by alternative large language models (LLMs).

The subsequent page, *NaC Sandbox Evaluator*, serves as a platform for calculating various metrics and applying the techniques described in Section 3.2.1. This functionality enables the derivation of a new dataset for each children dataset, incorporating the computed evaluation metrics.

Finally, the *NaC Evaluation Charts* page allows for the graphical representation of the evaluation results obtained from the previous page. This section presents both individual and aggregated statistical representations of performance metrics, facilitating effective comparative analyses across different LLMs.

The *NaC Dataset Explorer* page provides users with the functionality to view both the original question dataset intended for prompting the LLM in golden dataset generation, as well as the generated golden dataset itself. These datasets are rendered as Pandas DataFrames,¹⁰ ensuring cross-compatibility across various formats (see Figure 3.5). To examine the responses of a specific LLM in greater detail, users may select a particular conversation using a dedicated dropdown menu. Since each question in the dataset is unique, each is treated as a single-turn conversation, with no further interaction following the initial response. Conversations are displayed in a chat format with Markdown support, as illustrated in Figure 3.6.

The second functionality of this page allows for the generation of "children" datasets derived from the golden dataset. Users can select one or more LLMs to perform an inference job (Figure 3.7), which involves presenting the LLMs with user queries while utilizing a system prompt containing the relevant chunks extracted from the vector database, as described in previous sections. Each LLM configuration is specified in a configuration file in the following format:

Listing 3.2: Example LLM Configuration File

```
models:
  - model: "llama3"
    model_full_name: "Llama 3 70B"
    base_url: "http://localhost:11434/v1/"
    api_key: "none"
    temperature: 0.1
    max_tokens: 2048

  - model: "llama3.1"
    model_full_name: "Llama 3.1 8B"
    base_url: "http://localhost:11434/v1/"
    api_key: "none"
    temperature: 0.1
    max_tokens: 2048
```

Through this configuration, each LLM is tailored with essential parameters, including model type, base URL, API key, temperature setting, and token limitations, enabling flexible adjustments based on specific requirements. To ensure reproducibility across all experiments, a temperature setting of 0.1 was consistently applied. This low temperature value is intended to reduce randomness in the generated outputs, providing a more stable basis for evaluation and comparison.

3.2.1 NaC Sandbox Evaluator

The *NaC Sandbox Evaluator* page serves as the core evaluation platform, assessing each LLM against the golden dataset. An object-oriented class, *Metric*, was developed specifically for this purpose, enabling the application of various metrics by comparing a reference text to a prediction text.

In the context of Network as Code (NaC), only a subset of metrics from this class were selected, aligned with a structured "*"pillar evaluation"*" approach, illustrated in Figure 3.8.

The first pillar focuses on traditional NLP text similarity metrics, such as the *BERTScore* (available in both native Python and Hugging Face implementations). For the Hugging Face implementation, we

¹⁰<https://github.com/pandas-dev/pandas>

	Id	System Prompt	User Prompt	LLM Answer	Category	Execute code	Model Name
0	installation	// The assistant is special	How can I install NaC Py	```bash pip install network_a	Initialization	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
1	device identification	// The assistant is special	In which ways can I ider	**Parameter Name** **Detai	General concepts	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
2	scopes list	// The assistant is special	Can you tell me all the a	### Device Status scopes **Sc	General concepts	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
3	qod session list	// The assistant is special	Can you list all of the av	### Network as Code QoS prof	QoS	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
4	qod life cycle	// The assistant is special	Can you display the life	# QoD Session Life Cycle A QoD	QoS	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
5	nac localization capabilities	// The assistant is special	Which localization featu	**Feature** **Description**	Localization	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
6	location parameter info	// The assistant is special	What is the maximum a	The maximum age parameter i	Localization	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
7	location parameter default values	// The assistant is special	In case maximum age p	If the maximum age parameter	Localization	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
8	slice definition	// The assistant is special	What is a Specialized N	A Specialized Network, also kn	Specialized Networks	NO	meta-llama/Meta-Llama-3.1-70B-Instruc
9	slice id	// The assistant is special	Can you provide some c	A Network Identifier is a uniqu	Specialized Networks	NO	meta-llama/Meta-Llama-3.1-70B-Instruc

Figure 3.5: Golden dataset represented as Pandas Dataframe

Parameter	Description
device	Information about the device for which congestion data is requested or subscribed to.
id	It represents the subscription identifier.
subscription_expire_time	Expiration time for the congestion subscription (using ISO 8601 format)
notification_url	URL where notifications about congestion updates will be sent.
notification_auth_token (Optional)	Authentication token for accessing the notification URL.
maxNumberOfReports (Optional)	Specify an integer value for the maximum number of reports to be sent in the subscription.
start	Start timestamp for retrieving congestion data. It should be an ISO 8601 formatted date string or a date-time object as shown above.
end	End timestamp for retrieving congestion data. It should be an ISO 8601 formatted date string or a date-time object as shown above.

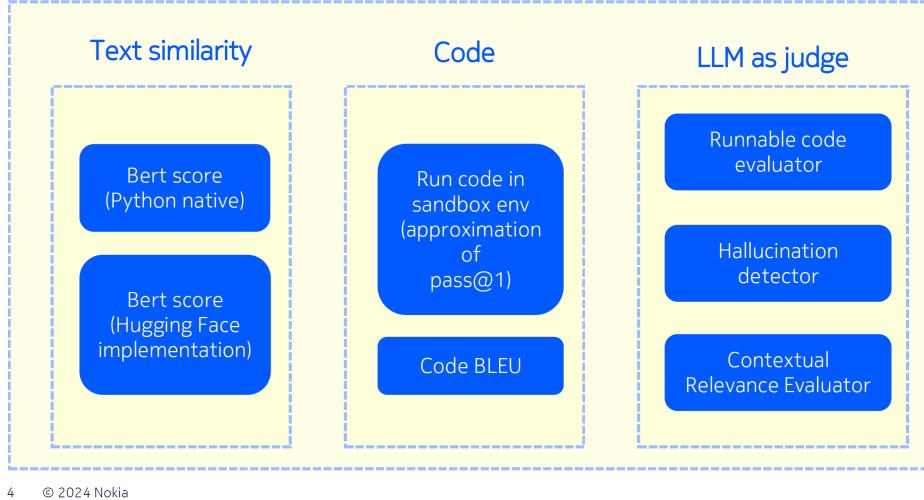
Figure 3.6: Example of user query and LLM answer represented as a single-turn chat

Figure 3.7: Section of the app where it is possible to generate children datasets from the golden one

used `distilbert-base-uncased_L5_no-idf_version=0.3.12(hug_trans=4.44.0)`, while for the native implementation, we used `roberta-large_L17_no-idf_version=0.3.0(hug_trans=2.3.0)`. This was done to evaluate whether changing the model would affect the results. However, as will be discussed in Section 4.1, both models exhibited similar behavior. The aim of this pillar is to conceptually assess the similarity between the evaluated and reference text. However, this pillar serves as a supplementary evaluation tool because, as explored in the *Background chapter* (Section 2), extensive research has shown that the value of such metrics has diminished over time. Newer methods, discussed

We want to structure the evaluation in easy to plug and play “pillars”

NaC evaluation



4 © 2024 Nokia



Figure 3.8: Proposed Pillars evaluation

in Section 2.4.4 have demonstrated greater effectiveness.

The second pillar, essential for this use case, emphasizes the quality of the code generated by the LLM. This pillar includes two components: a traditional metric and a pass/fail code test. The traditional metric used is *CodeBLEU* [145], which extends the capabilities of the standard BLEU metric [135] by introducing a *syntax match score* and *data-flow match score*. CodeBLEU evaluates not only lexical similarities but also considers syntactic and semantic accuracy, which is critical in assessing generated code. To achieve this, it computes a syntactic *Abstract Syntax Tree (AST)* match score, comparing the structural representation of the code—capturing its hierarchical, language-specific syntax. Additionally, it computes a semantic data-flow match score, which examines the logical flow of data within the code, thus assessing whether variable dependencies and transformations align with expected behavior. The formula for calculating CodeBLEU, shown in Figure 3.9, illustrates how these elements combine to produce a more nuanced and robust metric for code quality assessment. The second component of this pillar is a containerized environment^[11] known as the *Sandbox*^[12] which enables execution of the code snippets generated by the LLM in an isolated environment, separate from the application. The pipeline of the code evaluation is represented in Figure 3.10. In cases where the LLM generates an answer that includes code (limited to the 22 previously mentioned instances), the English text is separated from the code segment. Subsequently, the API key required for SDK usage is replaced (without LLM involvement). The script produced by the LLM is then executed within a sandbox environment, generating a dictionary that contains the execution output along with a flag indicating whether an error was encountered. Given that code generation and execution are time-intensive tasks, it was not feasible to conduct a *pass@k* test [113], as this would require generating and executing multiple instances of each snippet. Therefore, each snippet is executed only once, producing a single pass or fail result without unit testing. To maintain a clean development environment, particularly for remote servers managing network operations, “cleaning snippets” are executed after each code snippet test. This approach ensures that no residual data from the previous test interferes with subsequent evaluations.

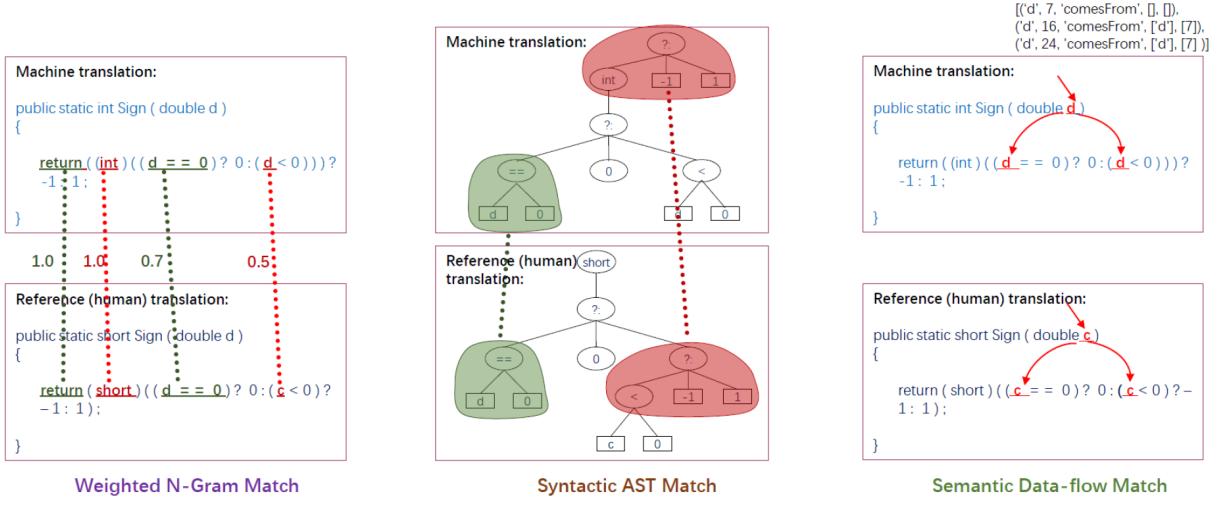
The third pillar employs the *LLM as Judge* methodology, noted for its versatility and effectiveness in recent studies [225] and [16]. This approach involves three specialized judges, inspired by the *Magis System*^[13] from the series *Neon Genesis Evangelion*^[14] each tasked with evaluating a distinct aspect of the LLM’s response.

^[11]<https://www.docker.com/resources/what-container/>

^[12]<https://github.com/vndee/llm-sandbox>

^[13]<https://wiki.evageeks.org/Magi>

^[14]https://en.wikipedia.org/wiki/Neon_Genesis_Evangelion



$$\text{CodeBLEU} = \alpha \cdot \text{N - Gram Match (BLEU)} + \beta \cdot \text{Weighted N-Gram Match} + \gamma \cdot \text{Syntactic AST Match} + \delta \cdot \text{Semantic Data-flow Match}$$

Figure 3.9: CodeBLEU metric components (Source: [145])

The first judge, referred to as the *Runnable Code Evaluator*, is responsible for assessing whether, given the provided context (chunks), the code snippet generated by the LLM will execute without errors. This judge returns a JSON object indicating the outcome, including the reasons for any anticipated errors.

The second judge, known as the *Hallucinator Detector*, evaluates whether the LLM's response is based solely on the retrieved context or if any part of the answer is a result of "hallucination."

Finally, the third judge, called the *Contextual Relevance Evaluator*, determines whether the LLM's response is aligned with the retrieved context. While this judge's task may seem similar to that of the Hallucinator Detector, it is crucial to note their distinct roles: the Hallucinator Detector aims to identify isolated instances of hallucination (such as a nonexistent code instruction), whereas the Contextual Relevance Evaluator performs a broader examination, verifying the groundedness of the LLM's response within the provided context.

The judges' approach is highly flexible, as each judge's configuration is specified in a YAML configuration file, shown in Listing 3.3. While, all metrics available in the app are shown in Figure 3.11.

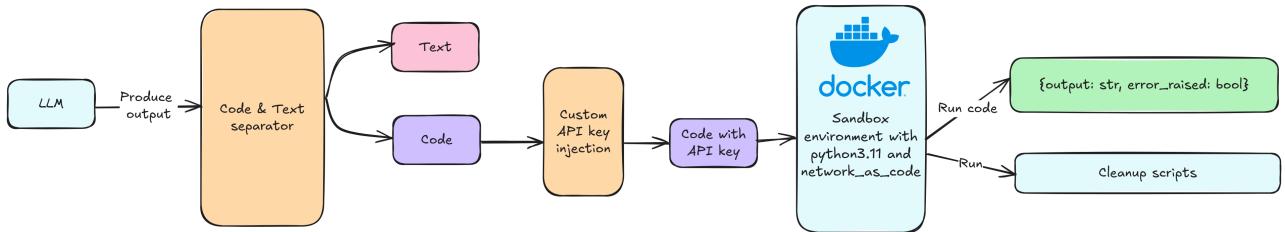


Figure 3.10: Code evaluation pipeline using Sandbox

Listing 3.3: Judges Configuration YAML file

```

judges:
  - judge_name: Runnable code evaluator
    instructions: |
      <instructions>
      You are an impartial judge that has to evaluate the correctness of a Large Language Model generated code.
      You have to understand the code and decide if the code will generate an error or not. Consider that the user will have already installed the network_as_code Python library. Please ignore any error that could be related to absence or wrong token.
      You are provided with some snippets of code retrieved from a vector database, the user query and the answer provided by the LLM.
      Please answer only with a Markdown JSON format containing a boolean value that indicated if the code generate errors and the reason why you took that decision.
      Here's an example:
      '''json
      {"error_will_be_raised": true, "reason": "An error will be raised because ..."}
      '''
      or
      '''json
      {"error_will_be_raised": false, "reason": "An error won't be raised because ..."}
      '''
      </instructions>
      ---
      Here's the context:
      <user_query>
      {user_query}
      </user_query>

      <context_retrieved>
      {context_retrieved}
      </context_retrieved>

      content_to_judge: |
        <llm_answer>
        {llm_answer}
        </llm_answer>
        ---
        Remember to only provide the answer in json format without any preamble or introductory text.

  - judge_name: Hallucination detector
    instructions: |
      <instructions>
      You are an impartial judge that has to evaluate if a Large Language Model provided any sort of hallucination while answering to user questions concerning the product Network as Code (NaC).
      Hallucination, in this context, refers to instances where the LLM provides information that is not present in the provided context or is entirely fabricated, and not based on any actual data or facts.
      You are provided with the context retrieved from a vector database containing the knowledge needed to answer the question, the user query and the answer provided by the LLM.
      Please note that in some cases, the LLM may provide additional code or instructions that are not explicitly mentioned in the context, but are necessary to complete an end-to-end solution. In such cases, these additions should not be considered as hallucinations.
      However, if the LLM provides information that contradicts the context or is entirely unrelated to the topic, it should be considered as a hallucination.
      Please answer only with a JSON format containing a boolean value that indicated if the

```

hallucination has been detected and the reason why you took that decision.
 Here's an example:
 '''json
 {"hallucination_detected": true, "reason": "..."}
 '''
 or
 '''json
 {"hallucination_detected": false, "reason": "..."}
 '''
<instructions>

Here's the context:
<user_query>
{user_query}
</user_query>

<context_retrieved>
{context_retrieved}
</context_retrieved>
 content_to_judge: |
<l1m_answer>
{l1m_answer}
</l1m_answer>

Remember to only provide the answer in json format without any preamble or introductory text.

- judge_name: Contextual Relevance Evaluator

instructions: |

You are an impartial judge that has to evaluate how well a Large Language Model generated response aligns with the provided context and user query concerning the product Network as Code (NaC).

You have to understand the response, the user query and the context retrieved from a vector database, and decide if the response addresses the specific problem or question asked.

The response can be either a text explanation or a code snippet.

Please answer only with a JSON format containing a boolean value that indicated if the response is contextually relevant and the reason why you took that decision.

Here's an example:

```
'''json
{"is_contextually_relevant": true, "reason": "..."}  

'''  

or  

'''json
{"is_contextually_relevant": false, "reason": "..."}  

'''  

</instructions>
```

Here's the context:
<user_query>
{user_query}
</user_query>

<context_retrieved>
{context_retrieved}
</context_retrieved>

content_to_judge: |
<l1m_answer>
{l1m_answer}

```
</llm_answer>
---
Remember to only provide the answer in json format without any preamble or
introductory text.
```

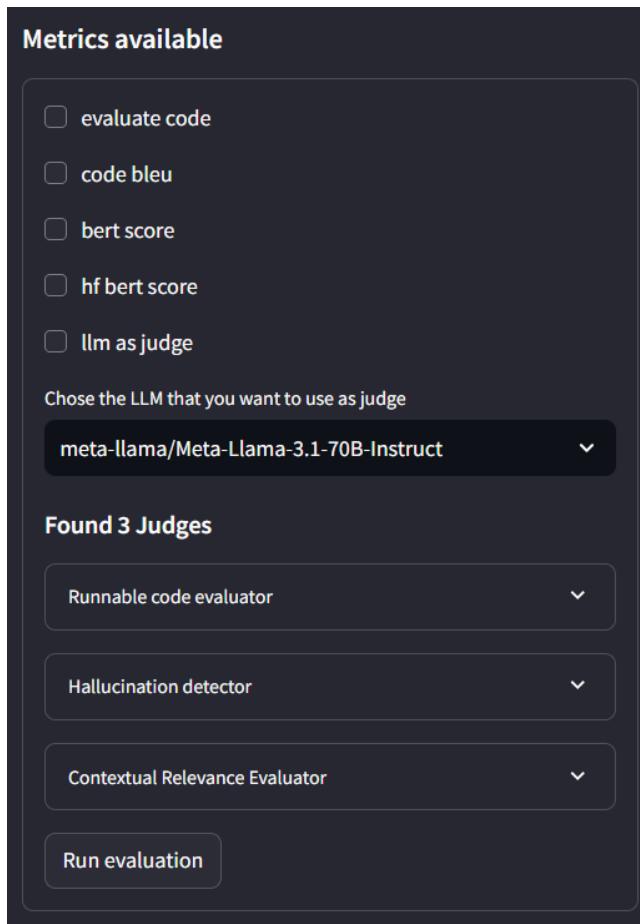


Figure 3.11: Available options for evaluation

3.2.2 NaC Evaluation Charts

The results obtained from the previous evaluation can be further analyzed in the *NaC Evaluation Charts* section, where users are enabled to review both individual LLM performance and aggregated statistics across models.

Each LLM has its dedicated report sheet, presenting high-level statistics that provide insights into the model's performance. These metrics include the number and percentage of code snippets that generated errors, the proportion of responses flagged as hallucinations by the judges, the accuracy rates of judges in predicting whether the code would produce errors, and the percentage of contextually relevant responses as evaluated by the contextual relevance judge (illustrated in Figure 3.12).

In addition to numerical statistics, radar plots are provided to break down various NLP metrics, such as Precision, Recall, and F1 Score, as shown in Figure 3.13.

For the aggregate statistics, all metrics and scores collected for each LLM are compiled, allowing users to further explore NLP metrics (*CodeBLEU*, *BERTScore*) for different prompt categories (see Figure 3.14). Additionally, radar plots illustrate model performance on the first evaluation pillar and a combined view for the second and third pillars, as shown in respectively in Figures 3.15 and 3.16.

To enable rapid model comparison, each metric has an associated bar chart comparing all models' performance (Figure 3.17).

Moreover, two heatmaps provide an overview of the evaluation scores across models for each evaluation pillar (Figures 3.18 and 3.19). Lastly, a correlation map illustrating relationships among LLM performance metrics is presented in Figure 3.20.



Figure 3.12: Scores for a single evaluated LLM (GPT 4o)

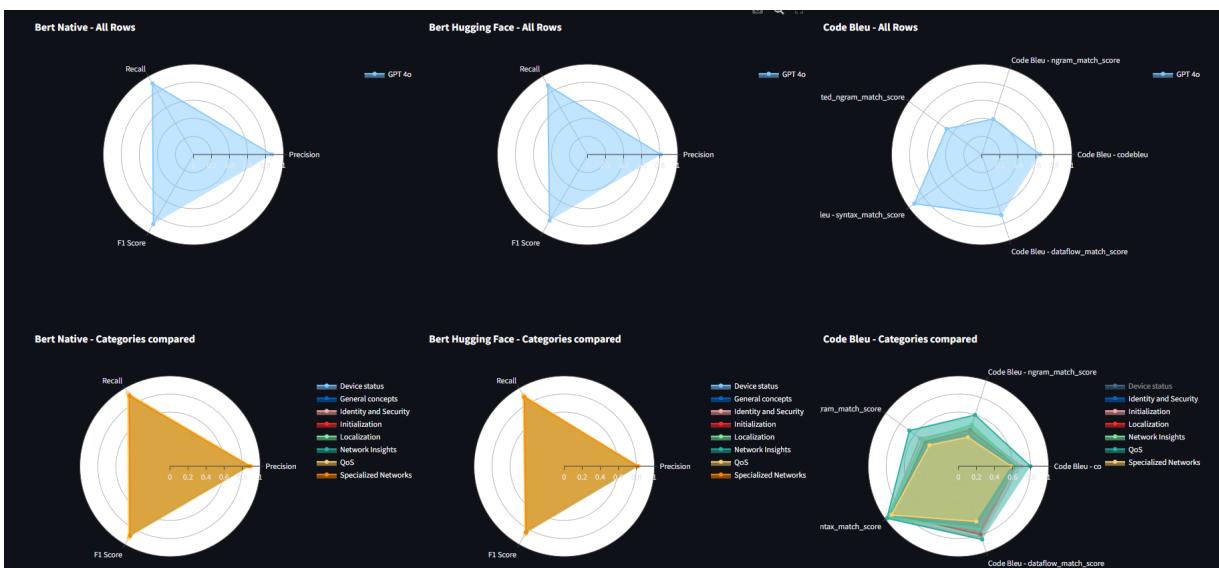


Figure 3.13: Radar plots for First pillar metrics of a single LLM

Model	↑ Code that raised errors (%)	Cases when judge detected hallucination (%)	Cases when judge decided that the content was contextual relevant (%)	Cases when judge correctly predicted that code would have generated errors (%)	Cases when judge incorrectly predicted (%)
GPT 4o	22.73	1.61	100	77.27	
Claude 3.5 Sonnet	31.82	0	98.39	63.64	
Qwen2.5 72.7B	40.91	0	100	59.09	
Mistral Nemo 12B	50	6.45	96.77	54.55	
Starcoder 2 16B	54.55	35.48	75.81	40.91	
DeepSeek-Coder-V2 16B	59.09	9.68	93.55	54.55	
GPT 4o mini	59.09	1.61	100	36.36	
Llama 3.1 8B	63.64	14.52	100	50	
Mistral 8x1B	63.64	14.52	85.48	45.45	
Qwen2.5 Coder 7.6B	63.64	1.61	98.39	36.36	

Figure 3.14: Aggregate statistics of each LLM scores

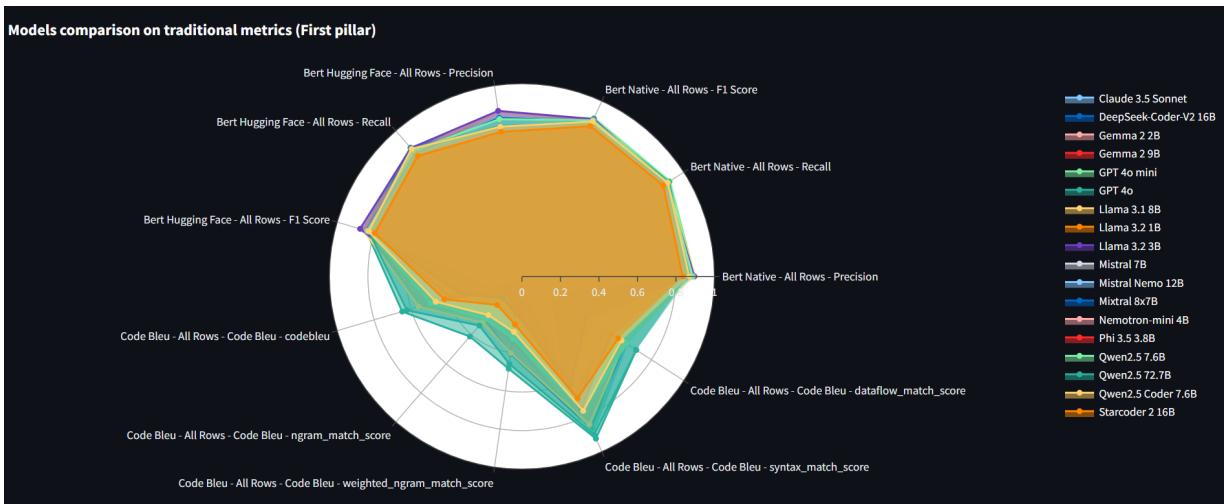


Figure 3.15: Radar plots for first pillar results

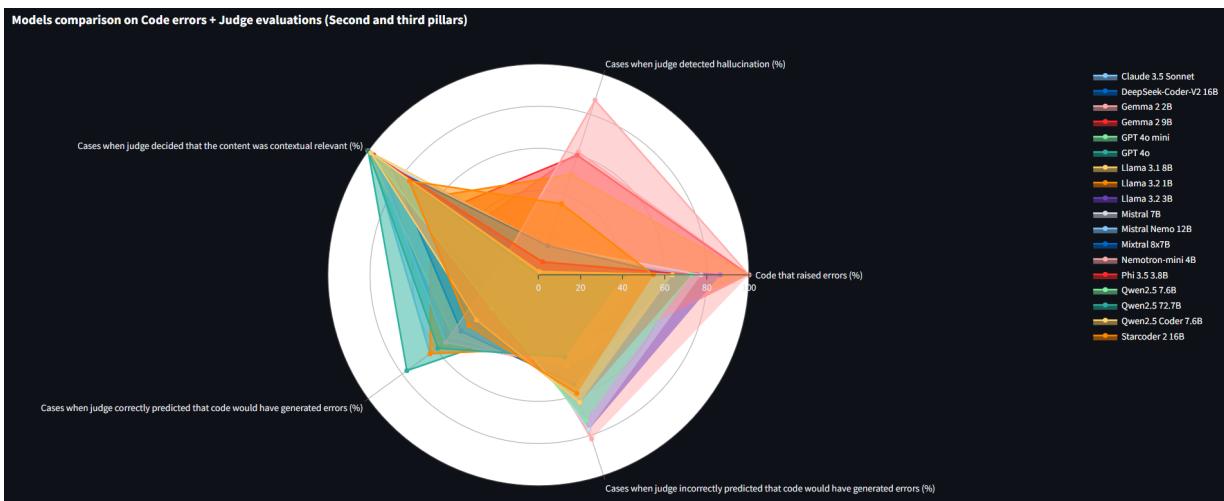


Figure 3.16: Radar plot for second and third pillar results

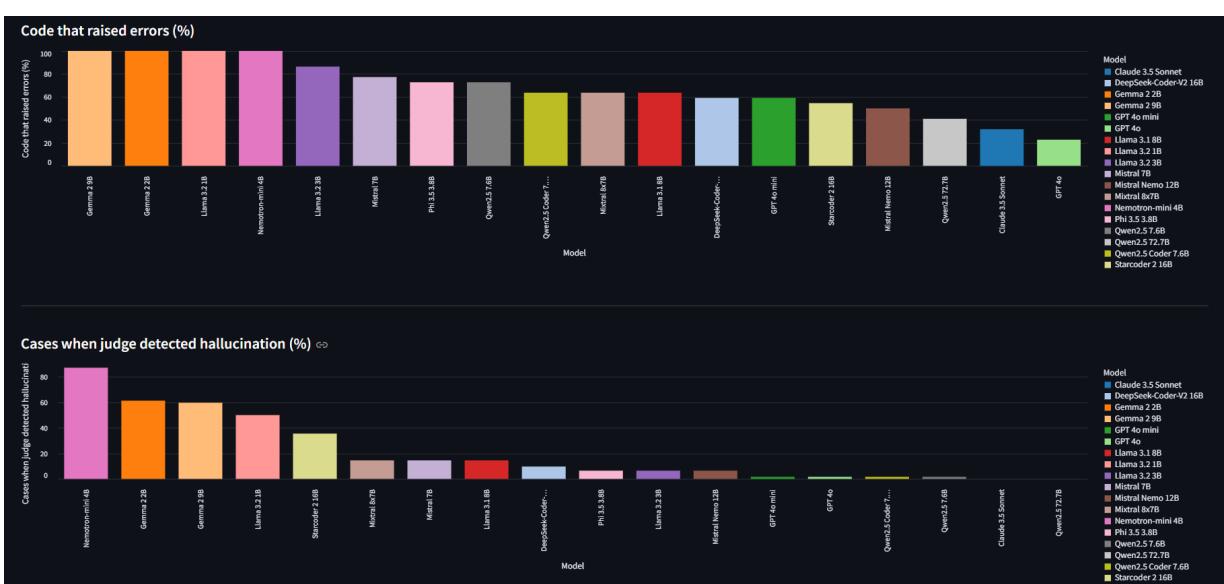


Figure 3.17: Bar chart displaying the LLMs that provided code that raised error, and LLMs that hallucinated in ascending order

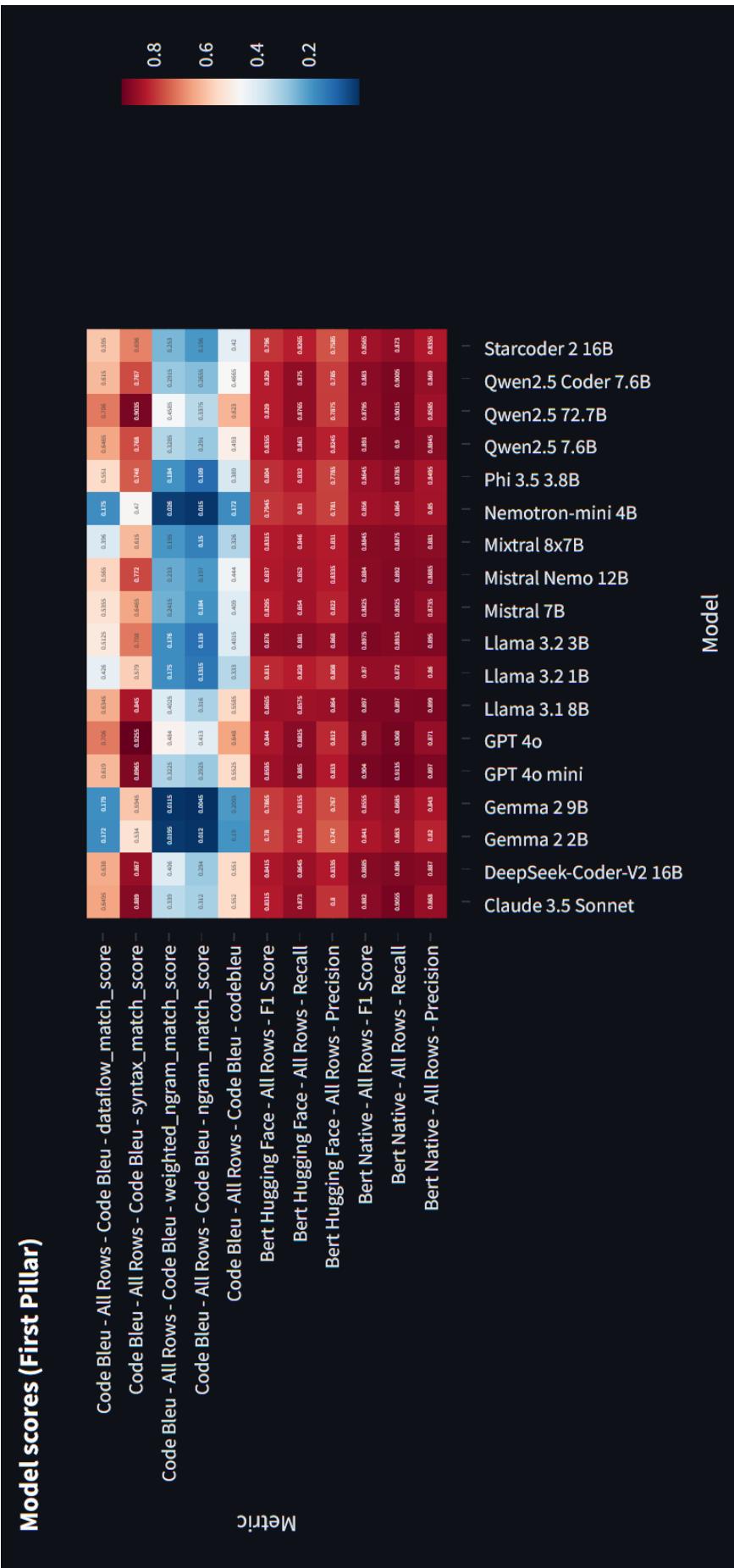


Figure 3.18: First pillar heatmap scores

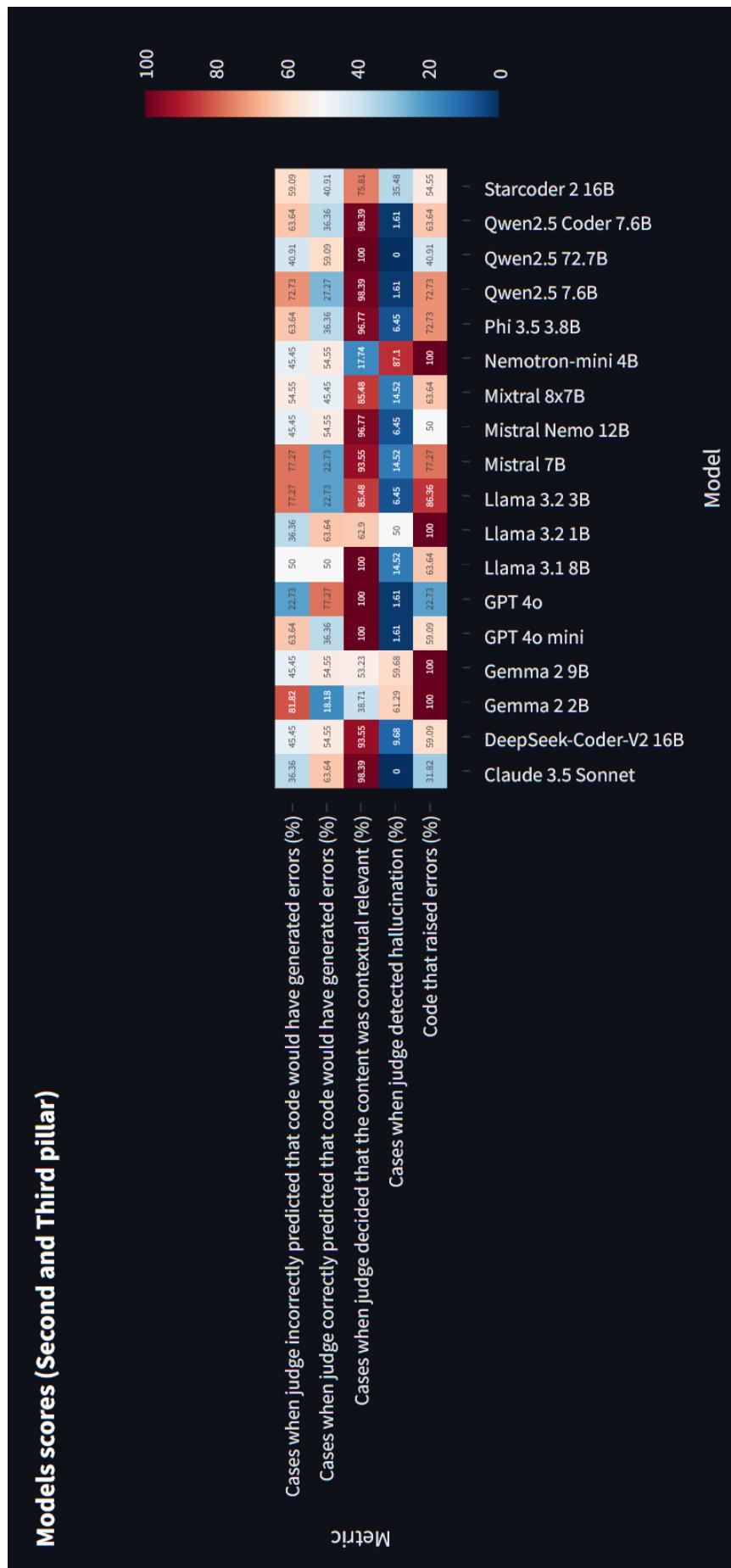


Figure 3.19: Second and third pillar heatmap scores

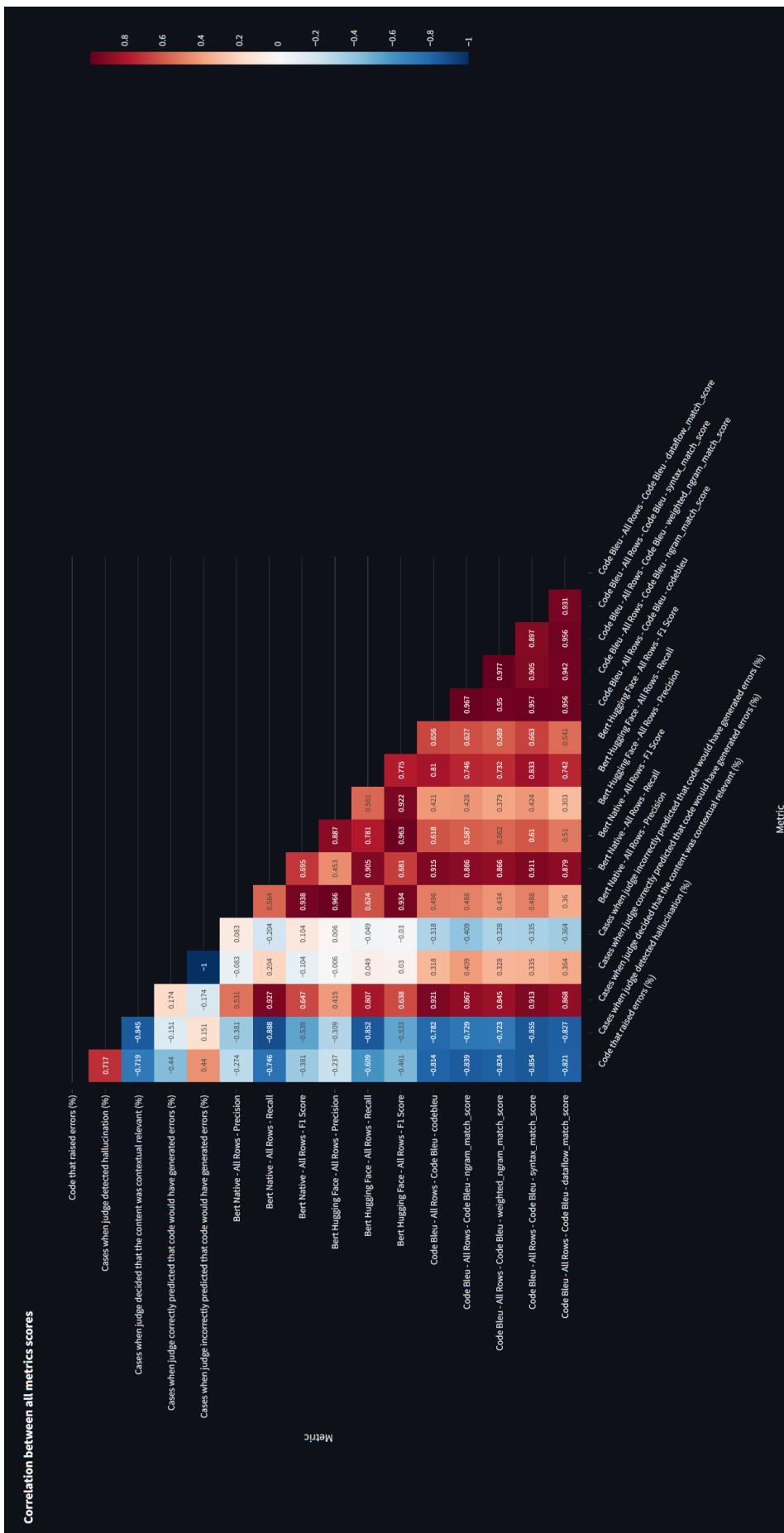


Figure 3.20: Correlation map between the metrics scores

4 Results

After constructing the complete pipeline, an experiment was conducted on a variety of LLMs, both open-source and closed-source. As outlined in the previous presentation, a golden dataset was first generated using *LLaMA 3.1 70B Instruct*, starting from a file containing the questions to be posed to each LLM. From this golden dataset, derivative (or "children") datasets were then created, which included only user queries and system prompts along with retrieved information from the vector database. This approach ensured a **consistent starting point** for all evaluated LLMs.

For this experiment, 18 LLMs were evaluated against the reference model, LLaMA 3.1 70B Instruct. The open-source models used in this evaluation include: *LLaMA 3.2 3B*, *LLaMA 3.2 1B*, *LLaMA 3.1 8B*, *Qwen 2.5 7.6B*, *Qwen 2.5-Coder 7.6B*, *Qwen 2.5 72B*, *Mixtral*, *Mistral*, *Mistral-Nemo*, *Nemotron-mini*, *Gemma2 2B*, *Gemma2 9B*, *Phi 3.5*, *Starcoder2 Instruct*, and *Deepseek-Coder-v2*. Details on each open-source LLM are provided in Table 4.1. These models were run using the Ollama platform.

The closed-source models used were *GPT-4o*, *GPT-4o-mini*, and *Claude-3.5 Sonnet*, which were run through the *Azure AI* platform.

Table 4.1: Specifications of Evaluated Open Source LLMs

Model Name	Number of Parameters	Context Length	Num Ctx	Embedding Length	Quantization Technique
llama3.2	3.0B	131072	16000	3072	Q4_K_M
llama3.2:1b	1.0B	131072	16000	2048	Q8_0
llama3.1	8.0B	131072	16000	4096	Q4_0
qwen2.5	7.6B	32768	16000	3584	Q4_K_M
qwen2.5:72b	72.7B	32768	16000	8192	Q4_K_M
qwen2.5-coder	7.6B	32768	16000	3584	Q4_K_M
mixtral	46.7B	32768	16000	4096	Q4_0
mistral	7.2B	32768	16000	4096	Q4_0
mistral-nemo	12.2B	131072	16000	5120	Q4_0
nemotron-mini	4.2B	4096	16000	3072	Q4_K_M
gemma2	9.2B	8192	8192	3584	Q4_0
gemma2:2b	2.6B	8192	8192	2304	Q4_0
phi3.5	3.8B	131072	16000	3072	Q8_0
starcoder2:instruct	16.0B	16384	16000	6144	Q4_0
deepseek-coder-v2	15.7B	163840	16000	2048	Q4_0

In this section, we will begin by discussing high-level aggregate results, providing an overview of overall performance trends across the evaluated models. Following this, we will offer a brief assessment of each individual model's performance. Subsequently, we will present a comparative analysis of the metrics, examining inter-metric correlations and relative strengths. Finally, we will explore the common patterns observed in the outputs generated by the LLMs, identifying recurring characteristics and notable trends.

4.1 Overview of Aggregate Performance Results

The results of the aggregated statistics reveal significant insights into the performance of various models in code generation tasks, as reported in Tables 4.2, 4.3, 4.4 and 4.5. The average code error rate across all models is notably high at 67.7%, with a standard deviation of 23.5%, indicating substantial

variability; the error rates range from 22.7% to 100%. This considerable discrepancy suggests that **many models face challenges in producing error-free code**.

Moreover, judges identified hallucinations in approximately 20.7% of the instances on average, with a wide variance indicated by a standard deviation of 26.4%. This variability is further emphasized by the fact that certain models reported no hallucinations, while others exhibited rates as high as 87%. Intriguingly, the detection of hallucinations appears to operate independently of the judges' predictive accuracy regarding code errors, as evidenced by an almost negligible correlation of 0.01 for correct predictions and -0.01 for incorrect predictions. This suggests that the ability to detect hallucinations is not necessarily linked to the accuracy of error predictions. The findings imply that **models with higher rates of hallucination detection do not inherently enhance the judges' predictive capabilities for code errors**. This independence may indicate a distinction where hallucination detection emphasizes semantic coherence, whereas error prediction is more aligned with the functionality of the code, resulting in minimal interaction between the two processes.

The mean accuracy of judges in correctly predicting outcomes stands at 45.5%, while the incorrect prediction rate is 54.5%, suggesting an **unpredictable behavior** akin to that of a coin toss. In terms of alignment, the **BERTScore** consistently exceeds 0.80, accompanied by a low standard deviation, **highlighting a robust overall text alignment**. The average score for the Native models is approximately 0.87, with a standard deviation of around 0.02, while the Hugging Face models display a slightly broader range from 0.81 to 0.85, with a standard deviation of 0.03. It is important to remind that the results from the two models are slightly different, as discussed in Section 3.2.1, due to the use of two distinct models. However, despite this difference, both models manifest the same behavior. Despite these high scores, it is noteworthy that several examples of working code received scores nearly identical to those of non-functional code, which underscores the **limitations of this metric as a reliable indicator of quality**.

Furthermore, the overall **CodeBLEU** scores may appear relatively low due to the equal weight assigned to all four components of the metric: *n-gram match*, *weighted n-gram match*, *syntax match*, and *dataflow match*. Specifically, the n-gram match and weighted n-gram match scores are relatively low, with both median and mean values converging around 0.20 for n-gram match and 0.25 for weighted n-gram match. In contrast, the syntax and dataflow scores are comparatively higher, with the syntax match exhibiting a mean of 0.74 and a median of 0.75, while the dataflow match shows a mean of 0.52 and a median of 0.58. This distinction indicates that the **models demonstrate greater consistency with syntax rather than the underlying data structures**.

Although this may appear counterintuitive, it represents a significant achievement given that adherence to syntax is crucial, especially when utilizing a SDK, as various pathways may lead to the desired outcome. Additionally, it is important to note that some LLMs followed the debugging instruction to "print to debug," which implies that models that did not adhere to this guidance may produce code with fewer instructions, thereby exhibiting a different data flow to achieve the same results.

Table 4.2: Error and Hallucination Statistics (Value range: [0.000,100.000])

Model	Scripts that raised errors (%)	Judge-detected hallucinations (%)	Judge-validated relevance (%)	Judge correct predictions (%)	cor-error (%)	Judge correct predictions (%)	incorrect error (%)
Mean	67.678	20.699	83.064	45.455		54.545	
Std	23.483	26.360	24.306	16.427		16.427	
Min	22.730	0.000	17.740	18.180		22.730	
25%	55.685	1.610	78.228	36.360		45.450	
50%	63.640	8.065	95.160	47.725		52.275	
75%	84.088	30.240	98.390	54.550		63.640	
Max	100.000	87.100	100.000	77.270		81.820	

Table 4.3: Bert Native Statistics (Value range: [0.000,1.000])

Model	Bert Native Precision	Bert Native Recall	Bert Native F1 Score
Mean	0.868	0.889	0.878
Std	0.022	0.016	0.017
Min	0.820	0.863	0.841
25%	0.852	0.874	0.866
50%	0.870	0.892	0.883
75%	0.886	0.900	0.889
Max	0.899	0.914	0.904

Table 4.4: Bert Hugging Face Statistics (Value range: [0.000,1.000])

Model	Bert HF Precision	Bert HF Recall	Bert HF F1 Score
Mean	0.807	0.852	0.826
Std	0.034	0.025	0.027
Min	0.747	0.81	0.78
25%	0.782	0.829	0.806
50%	0.81	0.856	0.830
75%	0.833	0.875	0.840
Max	0.868	0.885	0.876

Table 4.5: Code Bleu Statistics (Value range: [0.000,1.000])

Model	Code Bleu	Ngram Match	Weighted Ngram	Syntax Match Score	Dataflow Match Score
Mean	0.429	0.202	0.253	0.735	0.518
Std	0.144	0.121	0.143	0.139	0.178
Min	0.172	0.0045	0.0115	0.47	0.172
25%	0.347	0.122	0.178	0.623	0.448
50%	0.432	0.1965	0.24725	0.7575	0.58
75%	0.552	0.293625	0.336375	0.8615	0.637125
Max	0.648	0.413	0.484	0.9255	0.706

4.2 Performance Evaluation of Individual Models

The performance evaluation of each model reveals significant variances in their ability to generate accurate and relevant code. All the discussed results can be observed in Tables 4.6, 4.7, 4.8 and 4.9.

Table 4.6: Error and Hallucination Statistics (Value range: [0.00,100.00])

Model	Scripts raised (%)	that errors	Judge-detected hallucinations (%)	Judge-validated relevance (%)	Judge correct error predictions (%)	Judge incorrect error predictions (%)
Claude 3.5 Sonnet	31.82		0.00	98.39	63.64	36.36
DeepSeek-Coder-V2 16B	59.09		9.68	93.55	54.55	45.45

Table 4.6 – *Continued from previous page*

Model	Scripts raised (%)	that errors (%)	Judge-detected hallucinations (%)	Judge-validated relevance (%)	Judge correct error predictions (%)	Judge incorrect error predictions (%)
Gemma 2 2B	100.00		61.29	38.71	18.18	81.82
Gemma 2 9B	100.00		59.68	53.23	54.55	45.45
GPT 4o mini	59.09		1.61	100.00	36.36	63.64
GPT 4o	22.73		1.61	100.00	77.27	22.73
LLaMA 3.1 8B	63.64		14.52	100.00	50.00	50.00
LLaMA 3.2 1B	100.00		50.00	62.9	63.64	36.36
LLaMA 3.2 3B	86.36		6.45	85.48	22.73	77.27
Mistral 7B	77.27		14.52	93.55	22.73	77.27
Mistral Nemo 12B	50.00		6.45	96.77	54.55	45.45
Mixtral 8x7B	63.64		14.52	85.48	45.45	54.55
Nemotron-mini 4B	100.00		87.10	17.74	54.55	45.45
Phi 3.5 3.8B	72.73		6.45	96.77	36.36	63.64
Qwen2.5 7.6B	72.73		1.61	98.39	27.27	72.73
Qwen2.5 72.7B	40.91		0.00	100.00	59.09	40.91
Qwen2.5 Coder 7.6B	63.64		1.61	98.39	36.36	63.64
Starcoder 2 16B	54.55		35.48	75.81	40.91	59.09

Table 4.7: Bert Native Statistics (Value range: [0.000,1.000])

Model	Bert Precision	Native	Bert Native Recall	Bert Native F1 Score
Claude 3.5 Sonnet	0.868		0.906	0.882
DeepSeek-Coder-V2 16B	0.887		0.896	0.888
Gemma 2 2B	0.820		0.863	0.841
Gemma 2 9B	0.843		0.868	0.855
GPT 4o mini	0.897		0.914	0.904
GPT 4o	0.871		0.908	0.889
LLaMA 3.1 8B	0.899		0.897	0.897
LLaMA 3.2 1B	0.860		0.872	0.870
LLaMA 3.2 3B	0.895		0.892	0.898
Mistral 7B	0.873		0.893	0.883
Mistral Nemo 12B	0.888		0.892	0.884
Mixtral 8x7B	0.881		0.888	0.885
Nemotron-mini 4B	0.851		0.864	0.856
Phi 3.5 3.8B	0.849		0.878	0.864
Qwen2.5 7.6B	0.884		0.900	0.891
Qwen2.5 72.7B	0.859		0.902	0.879
Qwen2.5 Coder 7.6B	0.869		0.901	0.883
Starcoder 2 16B	0.835		0.873	0.857

Table 4.8: Bert Hugging Face Statistics (Value range: [0.000,1.000])

Model	Bert HF Precision	Bert HF Recall	Bert HF F1 Score
Claude 3.5 Sonnet	0.800	0.873	0.831
DeepSeek-Coder-V2 16B	0.833	0.865	0.841
Gemma 2 2B	0.747	0.818	0.780
Gemma 2 9B	0.767	0.815	0.787
GPT 4o mini	0.833	0.885	0.859
GPT 4o	0.812	0.883	0.844
LLaMA 3.1 8B	0.864	0.857	0.860
LLaMA 3.2 1B	0.808	0.828	0.811
LLaMA 3.2 3B	0.868	0.881	0.876
Mistral 7B	0.822	0.854	0.829
Mistral Nemo 12B	0.833	0.852	0.837
Mixtral 8x7B	0.831	0.846	0.831
Nemotron-mini 4B	0.781	0.810	0.795
Phi 3.5 3.8B	0.779	0.832	0.804
Qwen2.5 7.6B	0.825	0.863	0.835
Qwen2.5 72.7B	0.787	0.876	0.829
Qwen2.5 Coder 7.6B	0.785	0.875	0.829
Starcoder 2 16B	0.759	0.827	0.796

Table 4.9: Code Bleu Statistics (Value range: [0.000,1.000])

Model	Code Bleu	Ngram Match	Weighted Ngram	Syntax Match Score	Dataflow Match Score
Claude 3.5 Sonnet	0.552	0.312	0.339	0.889	0.649
DeepSeek-Coder-V2 16B	0.551	0.294	0.406	0.867	0.638
Gemma 2 2B	0.190	0.012	0.020	0.534	0.172
Gemma 2 9B	0.201	0.005	0.012	0.595	0.179
GPT 4o mini	0.553	0.293	0.323	0.897	0.619
GPT 4o	0.648	0.413	0.484	0.926	0.706
LLaMA 3.1 8B	0.559	0.316	0.403	0.845	0.635
LLaMA 3.2 1B	0.333	0.132	0.175	0.579	0.426
LLaMA 3.2 3B	0.402	0.119	0.176	0.708	0.513
Mistral 7B	0.409	0.184	0.242	0.647	0.536
Mistral Nemo 12B	0.444	0.197	0.233	0.772	0.565
Mixtral 8x7B	0.326	0.150	0.195	0.615	0.396
Nemotron-mini 4B	0.172	0.015	0.026	0.470	0.175
Phi 3.5 3.8B	0.804	0.389	0.109	0.748	0.551
Qwen2.5 7.6B	0.493	0.291	0.329	0.768	0.647
Qwen2.5 72.7B	0.623	0.338	0.459	0.904	0.706
Qwen2.5 Coder 7.6B	0.467	0.266	0.292	0.767	0.615
Starcoder 2 16B	0.420	0.196	0.253	0.696	0.595

Claude exhibited a script error rate of 31.82%. The judges did not identify any hallucinations, and only 1.6% of the examples were considered irrelevant based on the context retrieved from the LLM. Specific errors included a *HTTP error 422* related to unprocessable content during device information fetching, likely due to misformatted values in function parameters. The localization verification encountered a similar issue with error 422. Additionally, the deletion of roaming status

resulted in hallucinations associated with subscription deletions, which were not detected by the judges. Connectivity status checks and instant congestion level fetching also returned errors, specifically a 404 error when attempting to retrieve congestion levels immediately after subscription, highlighting a timing issue. The prediction of congestion levels faced analogous problems, indicating the model’s tendency to poll for information prematurely.

DeepSeekCoder v2 16B demonstrated a significantly higher script error rate of 59.09%. For instance, the “*hello world*” client initialization led to an unnecessary attempt at location verification, which generated error 422 due to incorrect handling. The device information fetching produced a hallucination error (*AttributeError: 'Device' object has no attribute 'get_info'*), which was not detected by the error code judge but was identified by the hallucination detector. Furthermore, the creation of QOD sessions with custom durations experienced hallucinations related to asynchronous calls, which both judges detected. The localization verification again attempted to perform additional computations, such as calculating the distance between *Rovaniemi* and another point, but failed because the required library was not installed. This behavior—characterized by the model’s tendency to add superfluous steps, seemingly to appear more sophisticated—resembled “the student who asks redundant questions to appear insightful,” yet ultimately resulted in more errors and unnecessary complexity. Errors persisted in device roaming status detection, and the model encountered an *AttributeError* related to non-existent attributes. The model’s attempts at fetching instant congestion levels and predicting congestion levels similarly resulted in errors, including a 404 error and an incorrect usage of the await keyword, respectively. A simulation of swapping verification for a custom number failed due to the use of an incorrect mobile number, resulting in an error 500.

Gemma 2B experienced a complete script error rate of 100%. In all examined code examples, issues predominantly arose from hallucinations related to import management. The model failed to adhere to instructions concerning the correct formatting of the application key. The judges displayed limited capability in determining the potential for code errors, particularly when an API key was not provided. The model’s syntax match score and dataflow scores were notably low, recorded at 0.53 and 0.17, respectively. Furthermore, the model often produced contextually irrelevant responses, diverging from the expected topics in 61% of cases or providing unnecessary instructions that frequently resulted in the generation of faulty code.

Gemma 9B, while an improvement over the 2B variant in terms of correctly including the application key string, still produced a 100% error rate. Similar to its predecessor, this model frequently hallucinated regarding import statements, which is a characteristic issue within this family of models. Additionally, it often failed to provide any code in specific swim swap scenarios.

GPT 4o mini yielded a 59.09% error rate in generated code. The model encountered challenges with `async` functions, primarily due to its lack of access to docstrings and code documentation, leading to incorrect type assumptions—such as treating a swap date as a datetime object when it was, in fact, a string. Errors in localization verification were unprocessable (error 422), largely because maximum age parameters were incorrectly formatted. Despite these coding errors, the contextual relevance of the generated code was generally satisfactory, with only 1.6% of outputs exhibiting hallucinations. This suggests that there is considerable potential for improvement, although these minor errors significantly impact code functionality.

GPT 4o emerged as the best-performing model in code generation, exhibiting the lowest error rate of 22.73% among the evaluated models. Among the errors encountered, one instance involved the model’s attempt to infer a function for deleting a subscription, which led to an incorrect output. The remaining errors appeared to stem from timing issues on the API side, suggesting that with a subsequent attempt, the generated code might be functional. Notably, the code generated by GPT 4o was sufficiently convincing that the judge, tasked with predicting functionality, incorrectly classified erroneous outputs as correct on five occasions. This finding highlights a notable limitation: the model’s apparent fluency can potentially mislead evaluators when the code’s surface accuracy obscures functional defects.

In contrast, **LLaMA 3.1 8B** exhibited a significantly higher error rate, with 63.64% of the scripts failing. The judge responsible for evaluating code reliability also showed a 50% error rate in predictions, suggesting challenges in discerning functionality. While LLaMA 3.1 8B maintained contextual

relevance across outputs, hallucinations were detected in 14.5% of cases. These hallucinations included the addition of extraneous information, such as methods absent from the provided context. Import errors were particularly prevalent in this model’s outputs, accompanied by issues with variable type misinterpretation and incorrect usage of `await` keyword. Such repeated import errors indicate potential deficiencies in the model’s ability to recognize standard library dependencies within code generation contexts.

The **LLaMA 3.2 1B** model, however, produced non-functional scripts in 100% of instances. The judge identified hallucinations in 50% of cases, while only 62.9% of outputs were deemed contextually relevant. This model exhibited a verbose tendency, often overcompensating by generating excessive tokens, resulting in erroneous outputs. One critical issue lay in the model’s inconsistent use of markdown formatting, making it difficult to extract usable Python code. Even in instances where the markdown structure was adequate, problems such as incorrect imports and mismatched parentheses persisted. Furthermore, the model’s lower BERTScores relative to its peers reinforce its relative underperformance across both contextual and technical dimensions.

The **LLaMA 3.2 3B** model demonstrated a relatively high error rate, with 86.36% of its scripts failing to execute correctly. This model effectively deceived the evaluation judge, who made only five accurate predictions regarding code errors, suggesting a notable gap between the code’s apparent functionality and its actual performance. A common source of errors involved the model’s inability to follow specific instructions, such as inserting placeholders like `<your-application-key-here>` into the designated locations within code, leading to the generation of unfunctional scripts. In other instances, the model exhibited mistakes in import syntax. Despite these issues, only 6.45% of the model’s responses were impacted by hallucinations, indicating that while the model demonstrated some consistency in content relevance, it frequently failed to maintain precise adherence to code structure and instructions. Interestingly, this model also exhibited a similar verbosity to its smaller counterpart, the 1B variant, often over-generating content beyond what was required by the prompt, which may have contributed to a misleading sense of code accuracy.

In comparison, **Mistral 7B** exhibited a slightly lower error rate of 77.27%. However, this model encountered notable issues with Python-specific formatting, as a substantial number of its errors were attributed to indentation misalignment—an essential aspect of Python code syntax. Further complicating its outputs, hallucinations occurred in 14.5% of responses, frequently manifesting as erroneous methods or imports that were not aligned with the prompt context. Despite these shortcomings, 93.5% of the responses were deemed contextually relevant, suggesting that while Mistral 7B’s outputs aligned with the intended task, technical execution fell short. Additionally, this model’s outputs often presented multiple Python scripts instead of a single focused response, complicating the task of extracting the correct code. The *BLEU* score analysis revealed lower results compared to other models, which likely reflects these structural issues and inconsistencies in handling complex instructions.

The **Mistral Nemo 12B** model, by contrast, achieved a relatively robust performance with a 50% error rate. This model excelled in producing contextually relevant responses, with 96.8% of its outputs aligning with prompt instructions, and exhibited a low hallucination rate of only 6.45%. Unlike its smaller counterparts, Mistral Nemo 12B generated notably precise and technically accurate code, though it struggled with asynchronous operations, particularly the correct use of `await` functions. Moreover, many of the model’s failures were due to server-side errors, such as 404 or 500 responses, implying that the model’s handling of parameters for network communication may have been flawed. This issue could potentially be mitigated by integrating a feedback loop, allowing the model to adjust its parameters dynamically based on real-time server responses. Interestingly, the model’s higher *n*-gram match scores in comparison to Mistral 7B further underscore its improved technical coherence, though data flow scores remained comparable. In 54.5% of cases, the judge assessed the generated code as error-free, a plausible outcome given that errors were often server-related rather than syntax-based, which may obscure issues for evaluators lacking specialized knowledge in NaC systems.

On the other hand, **Mixtral 8x7B** model displayed an intermediate error rate of 63.64%. This model encountered persistent issues with “forgetfulness”, as reflected in incomplete syntax: strings were often left without closing quotation marks, and the required placeholder token `<your-application-key-here>` was frequently omitted from designated fields. Additionally, hallucinations emerged in

the form of incorrect method suggestions for NaC operations, indicating an inconsistency in understanding contextual method usage. The model’s performance on *CodeBLEU* was notably weaker than that of the other Mixtral models (specifically Mistral 7B and Nemo 12B), pointing to a gap in both syntactical reliability and adherence to prompt constraints.

The **Nemotron-mini 4B** model demonstrated considerable limitations, generating scripts with a 100% error rate. This model encountered a wide array of issues that combined challenges observed in multiple other models, such as significant import errors, improper or missing use of the placeholder token `<your-application-key-here>`, and frequent omissions of essential Python keywords. Additionally, incorrect usage of asynchronous functions—particularly the `await` keyword—was pervasive. These combined issues suggest that Nemotron-mini 4B lacks a coherent understanding of Python’s structural and syntactic requirements. Compounding these technical flaws, the model exhibited a high hallucination rate of 87.1%, producing content that frequently deviated from the context established by the prompt. An alarming 82.3% of outputs were rated as not contextually relevant, underscoring the model’s apparent unreliability for code generation. This combination of hallucinations, context misalignment, and syntax errors renders Nemotron-mini 4B unsuitable for applications requiring consistent, reliable code.

Conversely, the **Phi 3.5 3.8B** model demonstrated somewhat better reliability, albeit with a high error rate of 72.73%. Unlike Nemotron-mini 4B, Phi 3.5 displayed a relatively low hallucination rate, with only 6.45% of outputs containing fabricated content and a mere 3.23% rated as contextually irrelevant. This suggests a certain degree of accuracy in understanding prompt requirements. However, despite its contextual alignment, Phi 3.5’s scripts frequently failed, primarily due to errors in handling NaC object attributes and function parameters, where the model often generated non-existent or incorrect methods and properties. While its outputs were frequently flawed, the syntax match score was relatively strong, indicating that Phi 3.5 generated syntactically valid Python code more consistently than some other models, including Mistral. Nonetheless, repeated import issues and hallucinated attribute usage point to substantial weaknesses in Phi 3.5’s underlying code logic and structure.

Qwen 2.5 7.6B exhibited an error rate of 72.73%, suggesting notable room for improvement in code reliability. Nevertheless, it achieved low rates of hallucination (1.61%) and contextually irrelevant content (1.61%), which speaks to a high level of adherence to prompt requirements. Despite these encouraging contextual metrics, the model struggled significantly with asynchronous programming, particularly in handling `await` and `async` functions, leading to frequent errors. Additionally, import-related hallucinations and misinterpretations of NaC object attributes contributed to its error rate. However, the model’s high syntax match and data flow scores indicate a strong grasp of syntactic structure and logical progression within its generated code, highlighting that the primary shortcomings are in specific functional areas rather than overall language comprehension.

On the other hand, **Qwen 2.5 Coder 7.6B** model showed a modest improvement over the standard 7.6B variant, with an error rate of 63.64%. Evaluation feedback was consistent with the non-Coder variant, with low hallucination and context-irrelevant content rates. Notably, many of the errors in Qwen 2.5 Coder 7.6B appeared to originate from server-side issues rather than from intrinsic syntactic or logical flaws. This model frequently generated code that was “nearly functional” but contained minor inaccuracies in function parameters or API calls that ultimately caused runtime errors. These findings suggest that the Qwen 2.5 Coder 7.6B model, while demonstrating reliable code generation capabilities, could benefit from an enhanced mechanism to handle API nuances or verify parameter accuracy, particularly in networked environments. *CodeBLEU* scores aligned closely with those of the Qwen 2.5 7.6B model, confirming that this variant retained the same structural and data flow strengths.

The **Qwen 2.5 72.7B** model achieved the lowest error rate among the Qwen series, with only 40.91% of scripts failing to execute correctly. Remarkably, the evaluating judges deemed all responses from this model contextually relevant and free from hallucinations, indicating consistent alignment with prompt objectives. This model scored exceptionally well in both syntax match (0.904) and data flow match (0.706) on the *CodeBLEU* metric, reflecting its superior understanding of code structure and flow. Despite this strong performance, some errors persisted, which may be indicative of nuanced

issues in parameter handling or subtle functional misunderstandings that did not compromise prompt relevance or syntactic correctness but still impacted functionality.

Turning to the **Starcoder 2 16B** model, this large language model exhibited a somewhat intermediate error rate of 54.55%, but significant issues were observed in the consistency and clarity of its outputs. Starcoder 2 frequently repeated identical code snippets and text sections, which resulted in unnecessarily verbose responses that complicated evaluation and usability. This repetition behavior suggests an over-generation tendency, potentially aimed at increasing the likelihood of generating accurate code within a single response. However, such redundancy led to increased rates of hallucination (35.5%) and context irrelevance (24.2%), as noted by the evaluating judges. Starcoder 2 also exhibited a broad range of common bugs, encompassing nearly all the issues observed in other models, yet with greater frequency, which further compromised its performance. Despite these issues, the model's syntax match and data flow match scores remained relatively high, though this might be artificially influenced by repeated code sections. For evaluation purposes, only the first code snippet was considered, but the redundancy suggests a lack of control in output management and a less refined approach to code generation.

Overall, the evaluated models exhibit varying capabilities in code generation, with **GPT 4o** emerging as the best performer, achieving a low error rate of 22.73% and demonstrating structural soundness alongside contextual relevance. **Claude 3.5 Sonnet** followed with an error rate of 31.82%, indicating good reliability with no detected hallucinations. **Qwen 2.5 72.7B** comes next with a 40.91% error rate, maintaining strong syntax and data flow scores but encountering minor functional issues, particularly in parameter handling. In contrast, models like **Gemma 2B**, **Gemma 9B**, **LLaMA 3.2 1B** and **Nemotron-mini 4B** displayed significant limitations, with error rates approaching 100% and high hallucination rates, undermining their reliability.

While models such as **Phi 3.5 3.8B** and **Qwen 2.5 7.6B** demonstrated fewer hallucinations and robust syntax matching, they struggled with asynchronous programming and function parameters. Furthermore, **Starcoder 2 16B** suffered from redundancy and verbosity, complicating output clarity. Overall, the analysis reveals that even the top-performing models face challenges with nuanced errors related to server interactions and asynchronous code, highlighting the need for improvements in parameter handling and code generation accuracy.

4.3 Comparative Analysis of Metrics

In this experiment, we analyze correlation scores using Spearman's rank correlation coefficient to evaluate the relationship between various metrics in code generation. The results reveal several notable patterns.

Firstly, we find a positive correlation of 0.717 between hallucination detection rates by the LLM judge and the percentage of scripts that raised an error. This suggests that if hallucinations are detected, there is a high likelihood of errors occurring in the code as well. This relationship is plausible because hallucinations often arise when there is misuse, misspelling, or incorrect application of code elements, leading to code errors.

A significant negative correlation of -0.845 exists between the judge-detected hallucinations and the perceived contextual relevance of the responses. This implies that an increase in hallucination detection is associated with a marked decrease in contextual relevance, as perceived by the judge. Therefore, higher hallucination rates appear to undermine the contextual appropriateness of the generated code.

The error rate in scripts exhibits a slight but negative correlation with the BERTScore. However, this correlation is not strong enough to conclude that higher BERTScores correspond to a lower frequency of buggy code. This outcome suggests that the BERTScore may not be a reliable metric for evaluating this specific type of code-related performance. Nevertheless, both the Native and Hugging Face implementations of the Recall metric show a strong positive correlation with cases where the judge validated the contextual relevance of the code.

Moreover, a high CodeBLEU score is strongly negatively correlated with the occurrence of errors in scripts. This implies that a higher CodeBLEU score is associated with a lower likelihood of bugs, suggesting that models achieving high CodeBLEU scores are likely to produce more reliable code. This finding indicates that a two-step evaluation process could be beneficial. Instead of immediately

executing code after obtaining metric scores, a CodeBLEU threshold could be set as a preliminary evaluation criterion. Only if the generated code meets this threshold would it proceed to the execution phase, potentially reducing evaluation time significantly.

Finally, as the gold-standard dataset contains only functional, factual examples, CodeBLEU is highly positively correlated with the judge’s validation of contextual relevance. Additionally, the CodeBLEU score is highly negatively correlated with cases in which the judge detected hallucinations, further reinforcing the robustness of CodeBLEU as an indicator of both contextual relevance and factual accuracy.

4.4 Common Patterns Observed in LLM Outputs

In examining common patterns across outputs from large language models (LLMs), several trends stand out. Firstly, nearly all LLMs struggled with generating asynchronous code reliably, likely due to limited access to complete SDK specifications, such as detailed Python docstrings for functions. Without this information, models frequently attempted to infer whether functions were asynchronous, often inserting the `await` keyword before functions without clear guidance.

Moreover, the error-detection mechanism, designed to assess whether scripts might produce errors, proved susceptible to certain oversights. While it can gauge syntactic correctness, it cannot fully ensure code functionality, since successful execution depends not only on code structure but also on interactions with server-side components that handle API requests. Issues like incorrect parameter passing or misinferred types sometimes led to hidden bugs, suggesting that this detection mechanism may not be entirely reliable.

Interestingly, the LLaMA model family achieved high BERTScores, which aligns with expectations, given that these models are distilled versions of larger counterparts that were used to generate the golden dataset, thus retaining similar token-selection capabilities.

We also observed that code-specialized LLMs tend to hallucinate more frequently, often providing excessive contextual information in a “divide-and-conquer” manner that ultimately yields suboptimal code.

Finally, while most LLMs generated well-structured code, subtle issues—such as incorrect parameters or API misuse—led to execution failures when run on remote servers. This contrast between apparent code quality and actual functionality highlights some persistent challenges in achieving robust LLM-generated code for practical applications.

5 Limitations and Future Works

In this work, we present a preliminary exploration into evaluating Large Language Models, acknowledging that both the pipeline and methodology employed here could be further refined and expanded. One notable limitation encountered was the restricted access to closed-source LLMs, such as Claude 3.5 Sonnet, GPT-4o, and GPT-4o mini, due to financial constraints imposed by AzureAI. The cost-per-inference model limited our token usage to a maximum of 10,000 tokens, which hindered our ability to use these models extensively for creating golden datasets or conducting broader experiments.

The quality of the LLM outputs could also be enhanced by incorporating the Network as Code SDK codebase, alongside the available documentation. Access to function docstrings could reduce or even eliminate errors related to asynchronous code, a frequent issue in this study. This problem partly arose because many models were forced to infer variable types and asynchronous function specifications in the absence of detailed documentation. The limited consistency in the documentation and the scarcity of examples contributed to this issue, making it difficult for the LLMs to generate accurate outputs consistently.

During the construction of the golden dataset, and even later in the evaluation phase using open-source LLMs, we could not fully utilize the LLMs' context windows due to resource limitations. Both hardware and computational resources were shared, necessitating constraints on character size per chunk to avoid long-running evaluations. Consequently, some LLMs with context windows as large as 132,000 tokens had fewer than 10% of this capacity utilized. However, we believe that the nature of the tasks and the simplicity of the scripts mitigated the impact of this token clipping on output quality.

Future research should explore various hyperparameter configurations within the RAG pipeline used to generate the golden dataset. Experimenting with these parameters would provide deeper insight into the pipeline's mechanisms and its key components, particularly in the retrieval stage. By defining reference questions, we could establish more precise metrics for retrieval performance. Such an undertaking would require close collaboration with specialized engineers involved in SDK development to accurately assess retrieval capabilities and optimize RAG components.

The importance of domain expertise has become increasingly evident through this evaluation. Although this initial study did not involve human evaluators, as it was designed to serve as a rapid proof of concept, consultation with engineers for SDK clarification revealed the need for consistent human oversight in future evaluations. While engineers were occasionally consulted for clarity, this limited involvement proved insufficient. Moving forward, the integration of domain experts will be critical for assessing the LLM outputs, given their ability to discern nuanced aspects of NaC-specific responses that general developers might overlook.

In subsequent iterations, it will be essential to involve human evaluators for comprehensive annotation and assessment of LLM responses within the NaC domain. This requirement is resource-intensive in terms of both cost and time, as engineers' primary responsibilities lie in SDK development, making this additional evaluation a significant task. Furthermore, incorporating A/B testing would allow human evaluators to compare responses from various LLMs without prior knowledge of the model used, thereby providing unbiased feedback. However, due to the domain's technical demands, these evaluations must be conducted by NaC-specialized engineers, as non-experts might be easily misled by model outputs.

Since the *LLM-as-Judge* approach has shown promise, future releases should consider incorporating additional specialized judges to enhance evaluation accuracy. Moreover, the outputs generated by these judges could serve as a feedback mechanism for the LLM itself, potentially enabling the model to iteratively refine and "correct" its own code responses, thereby improving the overall quality of the output.

Future iterations should also experiment further with prompt engineering. While prompt adjust-

ments were implemented in this study to address common LLM pitfalls (such as retaining device-specific configurations), these modifications often resulted in verbose prompts. A balance must be struck between prompt specificity and output quality to optimize responses. Streamlining prompts could contribute to improved LLM efficiency and accuracy.

Additionally, expanding hyperparameter exploration within the RAG pipeline remains a high priority, as our current approach focused solely on the LLM component itself. Further optimization in both the RAG pipeline and prompt engineering could enhance performance and reduce misinterpretations within specific documentation.

Finally, it is important to note that human evaluators were initially excluded due to cost considerations and the prioritization of engineering resources for SDK development. However, for future iterations, their involvement will be indispensable, especially in validating LLM-generated documentation. Technical details often require a thorough understanding of the NaC product, a level of expertise only achievable through direct involvement with specialized personnel. Thus, the inclusion of human evaluators is a requisite for the next phase of this work, ensuring more robust and accurate validation of LLM outputs.

6 Conclusion and discussion

This thesis has addressed the evaluation of Large Language Models (LLMs) with a focus on creating a tailored tool for assessing LLMs within specific business contexts. The developed tool specifically evaluates LLM performance within a Retrieval-Augmented Generation (RAG) pipeline, encompassing all stages from the generation of golden and child datasets to the evaluation and visualization of results. Central to the tool’s design is a comprehensive ”three pillars” evaluation framework, which includes (i) Text Similarity, (ii) Code Evaluation, and (iii) the LLM-as-Judge assessment. This triadic approach allows for a more holistic analysis of LLMs by examining language relevance, code quality, and judgment decisions. While this thesis applied the tool to a dataset focused on Network as Code developer SDK documentation, the tool’s modular and adaptable structure enables its application to diverse knowledge bases.

Through this work, Nokia now possesses an in-house proprietary tool for evaluating LLMs without reliance on external libraries. This tool offers substantial time savings, allowing engineers to quickly assess LLM performance through quantifiable metrics rather than subjective evaluation through direct interactions. These metrics provide a more objective basis for evaluation, although domain experts’ involvement will be essential for further iterations, ensuring domain-relevant insights guide assessment.

In our experiments, we evaluated 18 LLMs against a benchmark model, *LLaMA 3.1 70B Instruct*. Our findings indicated that the model *GPT-4o* demonstrated the lowest error rate in code generation, at 22.73%, followed by *Claude 3.5 Sonnet* (31.82%) and *Qwen 2.5 72.7B* (40.91%). Conversely, models like *Gemma 2B*, *Gemma 9B*, *LLaMA 3.2 1B*, and *Nemotron-mini 4B* exhibited a 100% error rate. These findings also confirmed previous research questioning the reliability of BERT Score as an evaluation metric. We observed that BERTScore often failed to detect nuanced differences in output quality; indeed, even the lowest-performing models achieved high BERTScores, with minimal distinction from the top-performing models.

A notable insight was that many LLMs encountered difficulties in generating asynchronous code. We posit that this limitation could be alleviated by expanding the RAG knowledge base to include the SDK codebase itself. Currently, only the SDK documentation is included, which leads to cases where LLMs have insufficient examples, prompting them to speculate on whether functions are asynchronous. Additionally, we validated the utility of CodeBLEU, especially for its syntax and dataflow match metrics, as a predictive metric for LLM code generation outcomes. A strong negative correlation was identified between CodeBLEU scores and the script error rates, underscoring CodeBLEU’s relevance as an evaluation tool for code-oriented tasks.

The study also highlighted the potential of the ”LLM-as-Judge” approach. While the judge model could not consistently predict if another LLM’s output was executable, the Hallucination Detector and Context Relevance Validator performed reliably. Nevertheless, introducing more human oversight in the evaluation loop would enhance accuracy. Positive correlations were found between instances where the judge detected hallucination and higher code error rates, while significant negative correlations emerged between hallucination rates and contextual relevance ratings.

Although LLM-generated code generally appeared well-structured, execution issues—especially in remote server interactions—remained prevalent, revealing a gap between code structure and real-world functionality. This underscores the ongoing challenge of practical applications for LLM-generated code, particularly in complex or environment-specific tasks. Additionally, specialized coding LLMs exhibited higher hallucination rates on custom knowledge bases when not exposed to the actual codebase. For example, models such as *DeepSeekCoder v2 16B* and *Starcoder 2 16B* often produced verbose, overly complex outputs in an attempt to compensate for their limited knowledge base. This tendency persisted even with prompts specifically instructing the model to generate concise solutions.

Future iterations of this work should include a comprehensive exploration of RAG hyperparameter tuning, alongside the testing of longer context windows to assess improvements in model performance.

Additionally, more thorough evaluations of closed-source models, limited in this iteration due to financial constraints and token usage caps, will provide further insights. Improvements in prompt engineering—particularly through shorter, more direct prompts—will enhance model usability. Refining the LLM-as-Judge approach and the instructions given to the evaluation model is also crucial. Finally, human evaluation by domain experts will be indispensable, filling the current gap in subjective assessments and ensuring rigorous validation of LLM outputs.

This research was made possible through the facilities, data, and support provided by Nokia Espoo (Finland), to whom I am deeply grateful.

Bibliography

- [1] Ebtesam Almazrouei et al. *The Falcon Series of Open Language Models*. 2023. arXiv: [2311.16867 \[cs.CL\]](#).
- [2] Alexandr Andoni, Piotr Indyk, and Ilya Razenshteyn. *Approximate Nearest Neighbor Search in High Dimensions*. 2018. arXiv: [1806.09823 \[cs.DS\]](#).
- [3] Daman Arora et al. “GAR-meets-RAG Paradigm for Zero-Shot Information Retrieval”. In: *arXiv preprint arXiv:2310.20158* (2023).
- [4] Akari Asai et al. “Self-RAG: Learning to Retrieve, Generate, and Critique through Self-Reflection”. In: *arXiv preprint arXiv:2310.11511* (2023).
- [5] Jinheon Baek et al. “Knowledge-Augmented Language Model Verification”. In: *arXiv preprint arXiv:2310.12836* (2023).
- [6] Yejin Bang et al. “A multitask, multilingual, multimodal evaluation of chatgpt on reasoning, hallucination, and interactivity”. In: *arXiv preprint arXiv:2302.04023* (2023).
- [7] Anja Belz and Ehud Reiter. “Comparing automatic and human evaluation of NLG systems”. In: *11th conference of the european chapter of the association for computational linguistics*. 2006, pp. 313–320.
- [8] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. “A neural probabilistic language model”. In: *J. Mach. Learn. Res.* 3.null (Mar. 2003), pp. 1137–1155.
- [9] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. “Semantic parsing on freebase from question-answer pairs”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1533–1544.
- [10] Moshe Berchansky et al. “Optimizing retrieval-augmented reader models via token elimination”. In: *arXiv preprint arXiv:2310.13682* (2023).
- [11] Sid Black et al. *GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow*. Version 1.0. If you use this software, please cite it using these metadata. Mar. 2021.
- [12] Sebastian Borgeaud et al. “Improving language models by retrieving from trillions of tokens”. In: *International conference on machine learning*. PMLR. 2022, pp. 2206–2240.
- [13] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: [2005.14165 \[cs.CL\]](#).
- [14] Sébastien Bubeck et al. “Sparks of artificial general intelligence: Early experiments with gpt-4”. In: *arXiv preprint arXiv:2303.12712* (2023).
- [15] Cristian Bucila, Rich Caruana, and Alexandru Niculescu-Mizil. *Model compression*. 2006.
- [16] Yupeng Chang et al. *A Survey on Evaluation of Large Language Models*. 2023. arXiv: [2307.03109 \[cs.CL\]](#).
- [17] Hung-Ting Chen, Fangyuan Xu, Shane A Arora, and Eunsol Choi. “Understanding retrieval augmentation for long-form question answering”. In: *arXiv preprint arXiv:2310.12150* (2023).
- [18] Jiang Chen. *Optimizing RAG with Rerankers: The Role and Trade-offs - Zilliz blog — zilliz.com*. <https://zilliz.com/learn/optimize-rag-with-rerankers-the-role-and-tradeoffs>. 2024.

- [19] Jiawei Chen, Hongyu Lin, Xianpei Han, and Le Sun. “Benchmarking large language models in retrieval-augmented generation”. In: *arXiv preprint arXiv:2309.01431* (2023).
- [20] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: [2107.03374 \[cs.LG\]](#).
- [21] Stanley F. Chen and Joshua Goodman. “An empirical study of smoothing techniques for language modeling”. In: *Computer Speech & Language* 13.4 (1999), pp. 359–394.
- [22] Tong Chen et al. “Dense X Retrieval: What Retrieval Granularity Should We Use?” In: *arXiv preprint arXiv:2312.06648* (2023).
- [23] Wenhui Chen, Xueguang Ma, Xinyi Wang, and William W Cohen. “Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks”. In: *arXiv preprint arXiv:2211.12588* (2022).
- [24] Daixuan Cheng et al. “UPRISE: Universal Prompt Retrieval for Improving Zero-Shot Evaluation”. In: *arXiv preprint arXiv:2303.08518* (2023).
- [25] Xin Cheng et al. “Lift Yourself Up: Retrieval-augmented Text Generation with Self Memory”. In: *arXiv preprint arXiv:2305.02437* (2023).
- [26] Yew Ken Chia et al. “Contrastive chain-of-thought prompting”. In: *arXiv preprint arXiv:2311.09277* (2023).
- [27] Wei-Lin Chiang et al. “Chatbot arena: An open platform for evaluating llms by human preference”. In: *arXiv preprint arXiv:2403.04132* (2024).
- [28] Wei-Lin Chiang et al. *Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality*. 2023.
- [29] Aeree Cho et al. “Transformer Explainer: Interactive Learning of Text-Generative Models”. In: *IEEE VIS* (2024).
- [30] Aakanksha Chowdhery et al. *PaLM: Scaling Language Modeling with Pathways*. 2022. arXiv: [2204.02311 \[cs.CL\]](#).
- [31] Peter Clark et al. “Think you have solved question answering? try arc, the ai2 reasoning challenge”. In: *arXiv preprint arXiv:1803.05457* (2018).
- [32] Karl Cobbe et al. “Training verifiers to solve math word problems”. In: *arXiv preprint arXiv:2110.14168* (2021).
- [33] Gordon V. Cormack, Charles L A Clarke, and Stefan Buettcher. “Reciprocal rank fusion outperforms condorcet and individual rank learning methods”. In: *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval. SIGIR ’09*. Boston, MA, USA: Association for Computing Machinery, 2009, pp. 758–759.
- [34] Florin Cuconasu et al. “The power of noise: Redefining retrieval for rag systems”. In: *arXiv preprint arXiv:2401.14887* (2024).
- [35] Pradeep Dasigi et al. “A dataset of information-seeking questions and answers anchored in research papers”. In: *arXiv preprint arXiv:2105.03011* (2021).
- [36] Yihe Deng, Weitong Zhang, Zixiang Chen, and Quanquan Gu. “Rephrase and respond: Let large language models ask better questions for themselves”. In: *arXiv preprint arXiv:2311.04205* (2023).
- [37] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: [1810.04805 \[cs.CL\]](#).
- [38] Shehzaad Dhuliawala et al. “Chain-of-verification reduces hallucination in large language models”. In: *arXiv preprint arXiv:2309.11495* (2023).
- [39] Shizhe Diao, Pengcheng Wang, Yong Lin, and Tong Zhang. “Active prompting with chain-of-thought for large language models”. In: *arXiv preprint arXiv:2302.12246* (2023).

- [40] Emily Dinan et al. “Wizard of wikipedia: Knowledge-powered conversational agents”. In: *arXiv preprint arXiv:1811.01241* (2018).
- [41] Xinya Du and Heng Ji. “Retrieval-Augmented Generative Question Answering for Event Argument Extraction”. In: *arXiv preprint arXiv:2211.07067* (2022).
- [42] Abhimanyu Dubey et al. *The Llama 3 Herd of Models*. 2024. arXiv: [2407.21783 \[cs.AI\]](https://arxiv.org/abs/2407.21783).
- [43] Seth Ebner et al. “Multi-sentence argument linking”. In: *arXiv preprint arXiv:1911.03766* (2019).
- [44] Hady Elsahar et al. “T-rex: A large scale alignment of natural language with knowledge base triples”. In: *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*. 2018.
- [45] Shahul Es, Jithin James, Luis Espinosa-Anke, and Steven Schockaert. “RAGAS: Automated Evaluation of Retrieval Augmented Generation”. In: *arXiv preprint arXiv:2309.15217* (2023).
- [46] Angela Fan et al. “ELI5: Long form question answering”. In: *arXiv preprint arXiv:1907.09190* (2019).
- [47] Zhangyin Feng et al. “Retrieval-Generation Synergy Augmented Large Language Models”. In: *arXiv preprint arXiv:2310.05149* (2023).
- [48] Clémentine Fourrier. 2023, year of open LLMs — *huggingface.co*. <https://huggingface.co/blog/2023-in-llms>, 2023.
- [49] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: [2312.10997 \[cs.CL\]](https://arxiv.org/abs/2312.10997).
- [50] Yonatan Geifman and Ran El-Yaniv. “Selective classification for deep neural networks”. In: *Advances in neural information processing systems* 30 (2017).
- [51] Mor Geva et al. “Did aristotle use a laptop? a question answering benchmark with implicit reasoning strategies”. In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 346–361.
- [52] Qi Gou et al. “Diversify Question Generation with Retrieval-Augmented Style Transfer”. In: *arXiv preprint arXiv:2310.14503* (2023).
- [53] Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q Weinberger. “On calibration of modern neural networks”. In: *International conference on machine learning*. PMLR. 2017, pp. 1321–1330.
- [54] Zhicheng Guo et al. “Prompt-Guided Retrieval Augmentation for Non-Knowledge-Intensive Tasks”. In: *arXiv preprint arXiv:2305.17653* (2023).
- [55] Moritz Hardt, Eric Price, and Nati Srebro. “Equality of opportunity in supervised learning”. In: *Advances in neural information processing systems* 29 (2016).
- [56] Hiroaki Hayashi et al. “WikiAsp: A dataset for multi-domain aspect-based summarization”. In: *Transactions of the Association for Computational Linguistics* 9 (2021), pp. 211–225.
- [57] Ruining He and Julian McAuley. “Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering”. In: *proceedings of the 25th international conference on world wide web*. 2016, pp. 507–517.
- [58] Xiaoxin He et al. “G-Retriever: Retrieval-Augmented Generation for Textual Graph Understanding and Question Answering”. In: *arXiv preprint arXiv:2402.07630* (2024).
- [59] Dan Hendrycks et al. “Measuring massive multitask language understanding”. In: *arXiv preprint arXiv:2009.03300* (2020).
- [60] Dan Hendrycks et al. “Measuring massive multitask language understanding”. In: *arXiv preprint arXiv:2009.03300* (2020).
- [61] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. *Distilling the Knowledge in a Neural Network*. 2015. arXiv: [1503.02531 \[stat.ML\]](https://arxiv.org/abs/1503.02531).

- [62] Xanh Ho, Anh-Khoa Duong Nguyen, Saku Sugawara, and Akiko Aizawa. “Constructing a multi-hop QA dataset for comprehensive evaluation of reasoning steps”. In: *arXiv preprint arXiv:2011.01060* (2020).
- [63] Jie Huang et al. “Raven: In-context learning with retrieval augmented encoder-decoder language models”. In: *arXiv preprint arXiv:2308.07922* (2023).
- [64] Hamel Husain et al. “Codesearchnet challenge: Evaluating the state of semantic code search”. In: *arXiv preprint arXiv:1909.09436* (2019).
- [65] Gautier Izacard et al. “Few-shot learning with retrieval augmented language models”. In: *arXiv preprint arXiv:2208.03299* (2022).
- [66] Neel Jain et al. *Bring Your Own Data! Self-Supervised Evaluation for Large Language Models*. 2023. arXiv: [2306.13651 \[cs.CL\]](#).
- [67] Palak Jain, Livio Baldini Soares, and Tom Kwiatkowski. “1-PAGER: One Pass Answer Generation and Evidence Retrieval”. In: *arXiv preprint arXiv:2310.16568* (2023).
- [68] Cheonsu Jeong. “Domain-specialized LLM: Financial fine-tuning and utilization method using Mistral 7B”. In: *Journal of Intelligence and Information Systems* 30.1 (Mar. 2024), pp. 93–120.
- [69] Jiaming Ji et al. “Beavertails: Towards improved safety alignment of llm via a human-preference dataset”. In: *Advances in Neural Information Processing Systems* 36 (2024).
- [70] Albert Q. Jiang et al. *Mixtral of Experts*. 2024. arXiv: [2401.04088 \[cs.LG\]](#).
- [71] Xinke Jiang et al. “Think and Retrieval: A Hypothesis Knowledge Graph Enhanced Medical Large Language Models”. In: *arXiv preprint arXiv:2312.15883* (2023).
- [72] Zhengbao Jiang et al. “Active retrieval augmented generation”. In: *arXiv preprint arXiv:2305.06983* (2023).
- [73] Bowen Jin et al. “Language Models As Semantic Indexers”. In: *arXiv preprint arXiv:2310.07815* (2023).
- [74] Karen Spärck Jones. “A statistical interpretation of term specificity and its application in retrieval”. In: *J. Documentation* 60 (2021), pp. 493–502.
- [75] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. “Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension”. In: *arXiv preprint arXiv:1705.03551* (2017).
- [76] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition with Language Models*. 3rd. Online manuscript released August 20, 2024. Pearson, 2024.
- [77] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: [2001.08361 \[cs.LG\]](#).
- [78] Zixuan Ke et al. “Bridging the Preference Gap between Retrievers and LLMs”. In: *arXiv preprint arXiv:2401.06954* (2024).
- [79] Omar Khattab et al. “Demonstrate-Search-Predict: Composing retrieval and language models for knowledge-intensive NLP”. In: *arXiv preprint arXiv:2212.14024* (2022).
- [80] Gangwoo Kim et al. “Tree of Clarifications: Answering Ambiguous Questions with Retrieval-Augmented Large Language Models”. In: *arXiv preprint arXiv:2310.14696* (2023).
- [81] Seungone Kim et al. “The CoT Collection: Improving Zero-shot and Few-shot Learning of Language Models via Chain-of-Thought Fine-Tuning”. In: *arXiv preprint arXiv:2305.14045* (2023).
- [82] Tomáš Kočiský et al. “The narrativeqa reading comprehension challenge”. In: *Transactions of the Association for Computational Linguistics* 6 (2018), pp. 317–328.
- [83] Neema Kotonya and Francesca Toni. “Explainable automated fact-checking for public health claims”. In: *arXiv preprint arXiv:2010.09926* (2020).

- [84] Tom Kwiatkowski et al. “Natural questions: a benchmark for question answering research”. In: *Transactions of the Association for Computational Linguistics* 7 (2019), pp. 453–466.
- [85] Tian Lan et al. “Copy is All You Need”. In: *The Eleventh International Conference on Learning Representations*. 2022.
- [86] Zhenzhong Lan et al. *ALBERT: A Lite BERT for Self-supervised Learning of Language Representations*. 2020. arXiv: [1909.11942 \[cs.CL\]](#).
- [87] Rémi Lebret, David Grangier, and Michael Auli. “Neural text generation from structured data with application to the biography domain”. In: *arXiv preprint arXiv:1603.07771* (2016).
- [88] Kuang-Huei Lee et al. “A Human-Inspired Reading Agent with Gist Memory of Very Long Contexts”. In: *arXiv preprint arXiv:2402.09727* (2024).
- [89] Noah Lee, Na Min An, and James Thorne. “Can large language models infer and disagree like humans”. In: *arXiv preprint arXiv:2305.13788* (2023).
- [90] Omer Levy, Minjoon Seo, Eunsol Choi, and Luke Zettlemoyer. “Zero-shot relation extraction via reading comprehension”. In: *arXiv preprint arXiv:1706.04115* (2017).
- [91] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: [2005.11401 \[cs.CL\]](#).
- [92] Patrick Lewis et al. “Retrieval-augmented generation for knowledge-intensive nlp tasks”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 9459–9474.
- [93] Cheng Li et al. “Large language models understand and can be enhanced by emotional stimuli”. In: *arXiv preprint arXiv:2307.11760* (2023).
- [94] Chengshu Li et al. “Chain of code: Reasoning with a language model-augmented code emulator”. In: *arXiv preprint arXiv:2312.04474* (2023).
- [95] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. “Structured chain-of-thought prompting for code generation”. In: *ACM Transactions on Software Engineering and Methodology* (2023).
- [96] Jian Li and Weiheng Lu. “A Survey on Benchmarks of Multimodal Large Language Models”. In: *arXiv preprint arXiv:2408.08632* (2024).
- [97] Sha Li, Heng Ji, and Jiawei Han. “Document-level event argument extraction by conditional generation”. In: *arXiv preprint arXiv:2104.05919* (2021).
- [98] Xiaoqian Li, Ercong Nie, and Sheng Liang. “From Classification to Generation: Insights into Crosslingual Retrieval Augmented ICL”. In: *arXiv preprint arXiv:2311.06595* (2023).
- [99] Xin Li and Dan Roth. “Learning question classifiers”. In: *COLING 2002: The 19th International Conference on Computational Linguistics*. 2002.
- [100] Xingxuan Li et al. “Chain-of-knowledge: Grounding large language models via dynamic knowledge adapting over heterogeneous sources”. In: *arXiv preprint arXiv:2305.13269* (2023).
- [101] Xinze Li et al. “Structure-Aware Language Model Pretraining Improves Dense Retrieval on Structured Data”. In: *arXiv preprint arXiv:2305.19912* (2023).
- [102] Percy Liang et al. *Holistic Evaluation of Language Models*. 2023. arXiv: [2211.09110 \[cs.CL\]](#).
- [103] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. In: *Text summarization branches out*. 2004, pp. 74–81.
- [104] Kevin Lin et al. *MPT: Mesh Pre-Training with Transformers for Human Pose and Mesh Reconstruction*. 2023. arXiv: [2211.13357 \[cs.CV\]](#).
- [105] Stephanie Lin, Jacob Hilton, and Owain Evans. “Truthfulqa: Measuring how models mimic human falsehoods”. In: *arXiv preprint arXiv:2109.07958* (2021).
- [106] Xi Victoria Lin et al. “RA-DIT: Retrieval-Augmented Dual Instruction Tuning”. In: *arXiv preprint arXiv:2310.01352* (2023).

- [107] Yen-Ting Lin and Yun-Nung Chen. *LLM-Eval: Unified Multi-Dimensional Automatic Evaluation for Open-Domain Conversations with Large Language Models*. 2023. arXiv: [2305.13711 \[cs.CL\]](#).
- [108] Ye Liu et al. “Exploring the integration strategies of retriever and large language models”. In: *arXiv preprint arXiv:2308.12574* (2023).
- [109] Yi Liu et al. “RECALL: A Benchmark for LLMs Robustness against External Counterfactual Knowledge”. In: *arXiv preprint arXiv:2311.08147* (2023).
- [110] Yinhan Liu et al. *RoBERTa: A Robustly Optimized BERT Pretraining Approach*. 2019. arXiv: [1907.11692 \[cs.CL\]](#).
- [111] Fan Luo and Mihai Surdeanu. “Divide & Conquer for Entailment-aware Multi-hop Evidence Retrieval”. In: *arXiv preprint arXiv:2311.02616* (2023).
- [112] Yuanjie Lyu et al. “CRUD-RAG: A comprehensive chinese benchmark for retrieval-augmented generation of large language models”. In: *arXiv preprint arXiv:2401.17043* (2024).
- [113] Zhi-Cun Lyu, Xin-Ye Li, Zheng Xie, and Ming Li. *Top Pass: Improve Code Generation by Pass@k-Maximized Code Ranking*. 2024. arXiv: [2408.05715 \[cs.AI\]](#).
- [114] Xinbei Ma et al. “Query Rewriting for Retrieval-Augmented Large Language Models”. In: *arXiv preprint arXiv:2305.14283* (2023).
- [115] Yubo Ma, Yixin Cao, YongChing Hong, and Aixin Sun. “Large language model is not a good few-shot information extractor, but a good reranker for hard samples!” In: *arXiv preprint arXiv:2303.08559* (2023).
- [116] Yu. A. Malkov and D. A. Yashunin. *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs*. 2018. arXiv: [1603.09320 \[cs.DS\]](#).
- [117] Alex Mallen et al. “When not to trust language models: Investigating effectiveness and limitations of parametric and non-parametric memories”. In: *arXiv preprint arXiv:2212.10511* (2022).
- [118] Potsawee Manakul, Adian Liusie, and Mark JF Gales. “Selfcheckgpt: Zero-resource black-box hallucination detection for generative large language models”. In: *arXiv preprint arXiv:2303.08896* (2023).
- [119] Nat McAleese et al. “Llm critics help catch llm bugs”. In: *arXiv preprint arXiv:2407.00215* (2024).
- [120] Eric Melz. “Enhancing llm intelligence with arm-rag: Auxiliary rationale memory for retrieval augmented generation”. In: *arXiv preprint arXiv:2311.04177* (2023).
- [121] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. “Pointer sentinel mixture models”. In: *arXiv preprint arXiv:1609.07843* (2016).
- [122] Shervin Minaee et al. *Large Language Models: A Survey*. 2024. arXiv: [2402.06196 \[cs.CL\]](#).
- [123] Timo Möller, Anthony Reina, Raghavan Jayakumar, and Malte Pietsch. “COVID-QA: A question answering dataset for COVID-19”. In: *ACL 2020 Workshop on Natural Language Processing for COVID-19 (NLP-COVID)*. 2020.
- [124] Subhabrata Mukherjee et al. *Orca: Progressive Learning from Complex Explanation Traces of GPT-4*. 2023. arXiv: [2306.02707 \[cs.CL\]](#).
- [125] Shashi Narayan, Shay B Cohen, and Mirella Lapata. “Don’t give me the details, just the summary! topic-aware convolutional neural networks for extreme summarization”. In: *arXiv preprint arXiv:1808.08745* (2018).
- [126] Tri Nguyen et al. “Ms marco: A human-generated machine reading comprehension dataset”. In: (2016).
- [127] Jekaterina Novikova, Ondřej Dušek, Amanda Cercas Curry, and Verena Rieser. “Why we need new evaluation metrics for NLG”. In: *arXiv preprint arXiv:1707.06875* (2017).

- [128] Maxwell Nye et al. “Show your work: Scratchpads for intermediate computation with language models”. In: *arXiv preprint arXiv:2112.00114* (2021).
- [129] Andrew M Olney. “Generating Multiple Choice Questions from a Textbook: LLMs Match Human Performance on Most Metrics.” In: *LLM@ AIED*. 2023, pp. 111–128.
- [130] OpenAI. *Introducing ChatGPT*. 2022.
- [131] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: [2303.08774 \[cs.CL\]](#).
- [132] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. *Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs*. 2024. arXiv: [2312.05934 \[cs.AI\]](#).
- [133] Oded Ovadia, Menachem Brief, Moshik Mishaeli, and Oren Elisha. “Fine-tuning or retrieval? comparing knowledge injection in llms”. In: *arXiv preprint arXiv:2312.05934* (2023).
- [134] Richard Yuanzhe Pang et al. “QuALITY: Question answering with long input texts, yes!” In: *arXiv preprint arXiv:2112.08608* (2021).
- [135] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. “Bleu: a method for automatic evaluation of machine translation”. In: *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 2002, pp. 311–318.
- [136] Bhargavi Paranjape et al. “Art: Automatic multi-step reasoning and tool-use for large language models”. In: *arXiv preprint arXiv:2303.09014* (2023).
- [137] Kaiping Peng, Richard E Nisbett, and Nancy YC Wong. “Validity problems comparing values across cultures and possible solutions.” In: *Psychological methods* 2.4 (1997), p. 329.
- [138] Chengwei Qin et al. “Is ChatGPT a general-purpose natural language processing task solver?” In: *arXiv preprint arXiv:2302.06476* (2023).
- [139] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. “Improving language understanding by generative pre-training”. In: *OpenAI blog* (2018).
- [140] Alec Radford et al. “Language models are unsupervised multitask learners”. In: *OpenAI blog* 1.8 (2019), p. 9.
- [141] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. “Squad: 100,000+ questions for machine comprehension of text”. In: *arXiv preprint arXiv:1606.05250* (2016).
- [142] Shashank Rajput et al. “Recommender Systems with Generative Retrieval”. In: *arXiv preprint arXiv:2305.05065* (2023).
- [143] Ori Ram et al. “In-context retrieval-augmented language models”. In: *arXiv preprint arXiv:2302.00083* (2023).
- [144] Ruiyang Ren et al. “Investigating the factual knowledge boundary of large language models with retrieval augmentation”. In: *arXiv preprint arXiv:2307.11019* (2023).
- [145] Shuo Ren et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. arXiv: [2009.10297 \[cs.SE\]](#).
- [146] Jon Saad-Falcon, Omar Khattab, Christopher Potts, and Matei Zaharia. “ARES: An Automated Evaluation Framework for Retrieval-Augmented Generation Systems”. In: *arXiv preprint arXiv:2311.09476* (2023).
- [147] Amrita Saha et al. “Complex sequential question answering: Towards learning to converse over linked question answer pairs with a knowledge graph”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.
- [148] Sourav Saha et al. “Vio-lens: A novel dataset of annotated social network posts leading to different forms of communal violence and its evaluation”. In: *Proceedings of the First Workshop on Bangla Language Processing (BLP-2023)*. 2023, pp. 72–84.
- [149] Pranab Sahoo et al. *A Systematic Survey of Prompt Engineering in Large Language Models: Techniques and Applications*. 2024. arXiv: [2402.07927 \[cs.AI\]](#).

- [150] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter*. 2020. arXiv: [1910.01108 \[cs.CL\]](https://arxiv.org/abs/1910.01108)
- [151] Parth Sarthi et al. “RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval”. In: *arXiv preprint arXiv:2401.18059* (2024).
- [152] Minju Seo, Jinheon Baek, James Thorne, and Sung Ju Hwang. “Retrieval-Augmented Data Augmentation for Low-Resource Domain Tasks”. In: *arXiv preprint arXiv:2402.13482* (2024).
- [153] C. E. Shannon. “Prediction and entropy of printed English”. In: *The Bell System Technical Journal* 30.1 (1951), pp. 50–64.
- [154] Zhihong Shao et al. “Enhancing Retrieval-Augmented Large Language Models with Iterative Retrieval-Generation Synergy”. In: *arXiv preprint arXiv:2305.15294* (2023).
- [155] Weizhou Shen et al. “Retrieval-Generation Alignment for End-to-End Task-Oriented Dialogue System”. In: *arXiv preprint arXiv:2310.08877* (2023).
- [156] Tianyuan Shi et al. “Dual-Feedback Knowledge Retrieval for Task-Oriented Dialogue Systems”. In: *arXiv preprint arXiv:2310.14528* (2023).
- [157] Weijia Shi et al. “Replug: Retrieval-augmented black-box language models”. In: *arXiv preprint arXiv:2301.12652* (2023).
- [158] Karan Singhal et al. “Large language models encode clinical knowledge”. In: *Nature* 620.7972 (2023), pp. 172–180.
- [159] Shamane Siriwardhana et al. “Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering”. In: *Transactions of the Association for Computational Linguistics* 11 (2023), pp. 1–17.
- [160] Richard Socher et al. “Recursive deep models for semantic compositionality over a sentiment treebank”. In: *Proceedings of the 2013 conference on empirical methods in natural language processing*. 2013, pp. 1631–1642.
- [161] Ralf Steinberger et al. “The JRC-Acquis: A multilingual aligned parallel corpus with 20+ languages”. In: *arXiv preprint cs/0609058* (2006).
- [162] Ivan Stelmakh, Yi Luan, Bhuwan Dhingra, and Ming-Wei Chang. “Asqa: Factoid questions meet long-form answers”. In: *arXiv preprint arXiv:2204.06092* (2022).
- [163] Zhiqing Sun et al. “Recitation-augmented language models”. In: *arXiv preprint arXiv:2210.01296* (2022).
- [164] Mirac Suzgun et al. “Challenging big-bench tasks and whether chain-of-thought can solve them”. In: *arXiv preprint arXiv:2210.09261* (2022).
- [165] Toni Taipalus. “Vector database management systems: Fundamental concepts, use-cases, and current challenges”. In: *Cognitive Systems Research* 85 (June 2024), p. 101216.
- [166] Alon Talmor, Jonathan Herzig, Nicholas Lourie, and Jonathan Berant. “Commonsenseqa: A question answering challenge targeting commonsense knowledge”. In: *arXiv preprint arXiv:1811.00937* (2018).
- [167] Rohan Taori et al. *Alpaca: A Strong, Replicable Instruction-Following Model*. <https://crfm.stanford.edu/2023/03/13/alpaca.html>. Accessed: 2024-10-13. 2023.
- [168] Gemma Team et al. *Gemma: Open Models Based on Gemini Research and Technology*. 2024. arXiv: [2403.08295 \[cs.CL\]](https://arxiv.org/abs/2403.08295)
- [169] Nandan Thakur et al. “NoMIRACL: Knowing When You Don’t Know for Robust Multilingual Retrieval-Augmented Generation”. In: *arXiv preprint arXiv:2312.11361* (2023).
- [170] James Thorne, Andreas Vlachos, Christos Christodoulopoulos, and Arpit Mittal. “FEVER: a large-scale dataset for fact extraction and VERification”. In: *arXiv preprint arXiv:1803.05355* (2018).

- [171] Katherine Tian et al. “Just ask for calibration: Strategies for eliciting calibrated confidence scores from language models fine-tuned with human feedback”. In: *arXiv preprint arXiv:2305.14975* (2023).
- [172] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: [2307.09288 \[cs.CL\]](https://arxiv.org/abs/2307.09288).
- [173] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. “Interleaving retrieval with chain-of-thought reasoning for knowledge-intensive multi-step questions”. In: *arXiv preprint arXiv:2212.10509* (2022).
- [174] Harsh Trivedi, Niranjan Balasubramanian, Tushar Khot, and Ashish Sabharwal. “MuSiQue: Multihop Questions via Single-hop Question Composition”. In: *Transactions of the Association for Computational Linguistics* 10 (2022), pp. 539–554.
- [175] Chris Van Der Lee et al. “Best practices for the human evaluation of automatically generated text”. In: *Proceedings of the 12th International Conference on Natural Language Generation*. 2019, pp. 355–368.
- [176] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: [1706.03762 \[cs.CL\]](https://arxiv.org/abs/1706.03762).
- [177] Kritin Vongthongsri. *LLM Benchmarks Explained: Everything on MMLU, HellaSwag, BBH, and Beyond - Confident AI — confident-ai.com*. <https://www.confident-ai.com/blog/llm-benchmarks-mmlu-hellaswag-and-beyond>. 2024.
- [178] Boxin Wang et al. “Adversarial glue: A multi-task benchmark for robustness evaluation of language models”. In: *arXiv preprint arXiv:2111.02840* (2021).
- [179] Boxin Wang et al. “DecodingTrust: A Comprehensive Assessment of Trustworthiness in GPT Models.” In: *NeurIPS*. 2023.
- [180] Boxin Wang et al. “Instructretro: Instruction tuning post retrieval-augmented pretraining”. In: *arXiv preprint arXiv:2310.07713* (2023).
- [181] Boxin Wang et al. “Shall we pretrain autoregressive language models with retrieval? a comprehensive study”. In: *arXiv preprint arXiv:2304.06762* (2023).
- [182] Cunxiang Wang et al. “Evaluating open question answering evaluation”. In: *arXiv preprint arXiv:2305.12421* (2023).
- [183] Hongru Wang et al. “Large Language Models as Source Planner for Personalized Knowledge-grounded Dialogue”. In: *arXiv preprint arXiv:2310.08840* (2023).
- [184] Hongru Wang et al. “Large Language Models as Source Planner for Personalized Knowledge-grounded Dialogue”. In: *arXiv preprint arXiv:2310.08840* (2023).
- [185] Hongru Wang et al. “UniMS-RAG: A Unified Multi-source Retrieval-Augmented Generation for Personalized Dialogue Systems”. In: *arXiv preprint arXiv:2401.13256* (2024).
- [186] Jindong Wang et al. “On the robustness of chatgpt: An adversarial and out-of-distribution perspective”. In: *arXiv preprint arXiv:2302.12095* (2023).
- [187] Liang Wang, Nan Yang, and Furu Wei. “Learning to retrieve in-context examples for large language models”. In: *arXiv preprint arXiv:2307.07164* (2023).
- [188] Xidong Wang et al. “Cmb: A comprehensive medical benchmark in chinese”. In: *arXiv preprint arXiv:2308.08833* (2023).
- [189] Xuezhi Wang et al. “Self-consistency improves chain of thought reasoning in language models”. In: *arXiv preprint arXiv:2203.11171* (2022).
- [190] Yidong Wang et al. “Pandalm: An automatic evaluation benchmark for llm instruction tuning optimization”. In: *arXiv preprint arXiv:2306.05087* (2023).
- [191] Yile Wang, Peng Li, Maosong Sun, and Yang Liu. “Self-Knowledge Guided Retrieval Augmentation for Large Language Models”. In: *arXiv preprint arXiv:2310.05002* (2023).

- [192] Yu Wang et al. “Knowledge graph prompting for multi-document question answering”. In: *arXiv preprint arXiv:2308.11730* (2023).
- [193] Yubo Wang et al. *MMLU-Pro: A More Robust and Challenging Multi-Task Language Understanding Benchmark (Published at NeurIPS 2024 Track Datasets and Benchmarks)*. 2024. arXiv: [2406.01574 \[cs.CL\]](#).
- [194] Zhenhailong Wang et al. “Zemi: Learning Zero-Shot Semi-Parametric Language Models from Multiple Tasks”. In: *arXiv preprint arXiv:2210.00185* (2022).
- [195] Zhiruo Wang et al. “Learning to filter context for retrieval-augmented generation”. In: *arXiv preprint arXiv:2311.08377* (2023).
- [196] Zilong Wang et al. “Chain-of-table: Evolving tables in the reasoning chain for table understanding”. In: *arXiv preprint arXiv:2401.04398* (2024).
- [197] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: [2201.11903 \[cs.CL\]](#).
- [198] Tsung-Hsien Wen et al. “Conditional generation and snapshot learning in neural dialogue systems”. In: *arXiv preprint arXiv:1606.03352* (2016).
- [199] BigScience Workshop et al. *BLOOM: A 176B-Parameter Open-Access Multilingual Language Model*. 2023. arXiv: [2211.05100 \[cs.CL\]](#).
- [200] Fangyuan Xu, Weijia Shi, and Eunsol Choi. “RECOMP: Improving Retrieval-Augmented LMs with Compression and Selective Augmentation”. In: *arXiv preprint arXiv:2310.04408* (2023).
- [201] Peng Xu et al. “Retrieval meets long context large language models”. In: *arXiv preprint arXiv:2310.03025* (2023).
- [202] Shicheng Xu et al. “Search-in-the-chain: Towards accurate, credible and traceable large language models for knowledgeintensive tasks”. In: *CoRR, vol. abs/2304.14732* (2023).
- [203] Xinchao Xu et al. “Long time no see! open-domain conversation with long-term persona memory”. In: *arXiv preprint arXiv:2203.05797* (2022).
- [204] Shi-Qi Yan, Jia-Chen Gu, Yun Zhu, and Zhen-Hua Ling. “Corrective Retrieval Augmented Generation”. In: *arXiv preprint arXiv:2401.15884* (2024).
- [205] An Yang et al. *Qwen2 Technical Report*. 2024. arXiv: [2407.10671 \[cs.CL\]](#).
- [206] Haoyan Yang et al. “PRCA: Fitting Black-Box Large Language Models for Retrieval Question Answering via Pluggable Reward-Driven Contextual Adapter”. In: *arXiv preprint arXiv:2310.18347* (2023).
- [207] Zhilin Yang et al. “HotpotQA: A dataset for diverse, explainable multi-hop question answering”. In: *arXiv preprint arXiv:1809.09600* (2018).
- [208] Shunyu Yao et al. “React: Synergizing reasoning and acting in language models”. In: *arXiv preprint arXiv:2210.03629* (2022).
- [209] Ori Yoran, Tomer Wolfson, Ori Ram, and Jonathan Berant. “Making retrieval-augmented language models robust to irrelevant context”. In: *arXiv preprint arXiv:2310.01558* (2023).
- [210] Wenhao Yu et al. “Chain-of-Note: Enhancing Robustness in Retrieval-Augmented Language Models”. In: *arXiv preprint arXiv:2311.09210* (2023).
- [211] Wenhao Yu et al. “Chain-of-note: Enhancing robustness in retrieval-augmented language models”. In: *arXiv preprint arXiv:2311.09210* (2023).
- [212] Wenhao Yu et al. “Generate rather than retrieve: Large language models are strong context generators”. In: *arXiv preprint arXiv:2209.10063* (2022).
- [213] Zichun Yu, Chenyan Xiong, Shi Yu, and Zhiyuan Liu. “Augmentation-Adapted Retriever Improves Generalization of Language Models as Generic Plug-In”. In: *arXiv preprint arXiv:2305.17331* (2023).

- [214] Rowan Zellers et al. “Hellaswag: Can a machine really finish your sentence?” In: *arXiv preprint arXiv:1905.07830* (2019).
- [215] Rowan Zellers et al. “Hellaswag: Can a machine really finish your sentence?” In: *arXiv preprint arXiv:1905.07830* (2019).
- [216] Rich Zemel et al. “Learning fair representations”. In: *International conference on machine learning*. PMLR. 2013, pp. 325–333.
- [217] Aohan Zeng et al. *GLM-130B: An Open Bilingual Pre-trained Model*. 2023. arXiv: [2210.02414 \[cs.CL\]](#).
- [218] Hui Zeng. “Measuring massive multitask chinese understanding”. In: *arXiv preprint arXiv:2304.12986* (2023).
- [219] Susan Zhang et al. *OPT: Open Pre-trained Transformer Language Models*. 2022. arXiv: [2205.01068 \[cs.CL\]](#).
- [220] Tianyi Zhang et al. “Bertscore: Evaluating text generation with bert”. In: *arXiv preprint arXiv:1904.09675* (2019).
- [221] Zhebin Zhang et al. “IAG: Induction-Augmented Generation Framework for Answering Reasoning Questions”. In: *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2023, pp. 1–14.
- [222] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. *Automatic Chain of Thought Prompting in Large Language Models*. 2022. arXiv: [2210.03493 \[cs.CL\]](#).
- [223] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2024. arXiv: [2303.18223 \[cs.CL\]](#).
- [224] Huaixiu Steven Zheng et al. “Take a step back: Evoking reasoning via abstraction in large language models”. In: *arXiv preprint arXiv:2310.06117* (2023).
- [225] Lianmin Zheng et al. *Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena*. 2023. arXiv: [2306.05685 \[cs.CL\]](#).
- [226] Lianmin Zheng et al. “Lmsys-chat-1m: A large-scale real-world llm conversation dataset”. In: *arXiv preprint arXiv:2309.11998* (2023).
- [227] Ming Zhong et al. “QMSum: A new benchmark for query-based multi-domain meeting summarization”. In: *arXiv preprint arXiv:2104.05938* (2021).
- [228] Ming Zhong et al. “Towards a unified multi-dimensional evaluator for text generation”. In: *arXiv preprint arXiv:2210.07197* (2022).
- [229] Yongchao Zhou et al. “Large language models are human-level prompt engineers”. In: *arXiv preprint arXiv:2211.01910* (2022).
- [230] Kaijie Zhu et al. “Promptbench: Towards evaluating the robustness of large language models on adversarial prompts”. In: *arXiv preprint arXiv:2306.04528* (2023).
- [231] Caleb Ziems et al. “Can large language models transform computational social science?” In: *Computational Linguistics* 50.1 (2024), pp. 237–291.

List of Acronyms

AI	Artificial Intelligence
ANN	Approximate Nearest Neighbor
ASR	Attack Success Rate
AUC	Area Under the Curve
BERT	Bidirectional Encoder Representations from Transformers
BLOOM	BigScience Large Open-science Open-access Multilingual Language Model
CoT	Chain of Thought
DPD	Demographic Parity Difference
ECE	Expected Calibration Error
EM	Exact Match
EOD	Equalized Odds Difference
FT	Fine-tuning
GPT	Generative Pre-Training
HNSW	Hierarchical Navigable Small World
IE	Information Extraction
LLaMA	Large Language Model Meta AI
LLM	Large Language Model
MLM	Masked Language Modeling
MLP	Multilayer perceptron
MMLU	Massive Multitask Language Understanding
NLG	Natural Language Generation
NLM	Neural Language Model
NLP	Natural Language Understanding
NLP	Natural Language processing
NSP	Next Sentence Prediction
OPT	Open Pre-trained Transformer
PaLM	Pathways Language Model

PDR	Performance Drop Rate
PLM	Pre-trained Language Model
RAG	Retrieval-Augmented Generation
RLHF	Reinforcement Learning from Human Feedback
RRF	Reciprocal Rank Fusion
SLM	Statistical Language Model
STEM	Science, Technology, Engineering and Mathematics
TF-IDF	Term Frequency-Inverse Document Frequency
VDBMS	Vector Database Management Systems
WER	Word Error Rate
IoT	Internet of Things
NLMs	Neural Language Models
PLMs	Pre-trained Language Models
QoD	Quality-of-Service on Demand
VR	Virtual Reality
XR	Extended Reality