A Hybrid RAG System with Comprehensive Enhancement on Complex Reasoning[*]

Ye Yuan[†]
State Key Laboratory of Multimedia Information Processing
School of Computer Science
PKU-Anker Embodied AI Lab
Peking University
100871, Beijing, China
yuanye_pku@pku.edu.cn

Chengwu Liu
State Key Laboratory of Multimedia Information Processing
School of Computer Science
PKU-Anker Embodied AI Lab
Peking University
100871, Beijing, China
liuchengwu@pku.edu.cn

Jingyang Yuan
State Key Laboratory of Multimedia Information Processing
School of Computer Science
PKU-Anker Embodied AI Lab
Peking University
100871, Beijing, China
yuanjy@pku.edu.cn

Gongbo Sun[‡]
School of Computer, Data & Information Sciences
University of Wisconsin-Madison
Madison, WI, United States
gsun43@wisc.edu

Siqi Li
State Key Laboratory of Multimedia Information Processing
School of Computer Science
PKU-Anker Embodied AI Lab
Peking University
100871, Beijing, China
xiaolilsq@stu.pku.edu.cn

Ming Zhang[§]
State Key Laboratory of Multimedia Information Processing
School of Computer Science
PKU-Anker Embodied AI Lab
Peking University
100871, Beijing, China
mzhang_cs@pku.edu.cn

Abstract

Retrieval-augmented generation (RAG) is a framework enabling large language models (LLMs) to enhance their accuracy and reduce hallucinations by integrating external knowledge bases. In this paper, we introduce a hybrid RAG system enhanced through a comprehensive suite of optimizations that significantly improve retrieval quality, augment reasoning capabilities, and refine numerical computation ability. We refined the text chunks and tables in web pages, added attribute predictors to reduce hallucinations, conducted LLM Knowledge Extractor and Knowledge Graph Extractor, and finally built a reasoning strategy with all the references. We evaluated our system on the CRAG dataset through the Meta CRAG KDD Cup 2024 Competition. Both the local and online evaluations demonstrate that our system significantly enhances complex reasoning capabilities. In local evaluations, we have significantly improved accuracy and reduced error rates compared to the baseline model, achieving a notable increase in scores. In the meanwhile, we have attained outstanding results in online assessments, demonstrating the performance and generalization capabilities of the proposed system. The source code for our system is released in https://gitlab.aicrowd.com/shizueyy/crag-new.

CCS Concepts

• **Computing methodologies → Natural language generation**; **Information extraction**.

Keywords

Large Language Models, Language Generation, Retrieval-Augmented Generation, Reasoning

ACM Reference Format:
Ye Yuan, Chengwu Liu, Jingyang Yuan, Gongbo Sun, Siqi Li, and Ming Zhang. 2024. A Hybrid RAG System with Comprehensive Enhancement on Complex Reasoning. In *KDDCup Workshop of SIGKDD '24: KDDCup Workshop of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 13 pages.

1 Introduction

Pre-trained large language models (LLMs), such as Llama3 [9], GPT-4 [60], Mistral-7B [24], Gemini family [59] and Qwen-2 [69], have demonstrated impressive advancements in the field of Natural Language Processing (NLP), notably in the Question-Answering task. This success is built on foundational knowledge, including factual knowledge [45], relational knowledge [50], and linguistic knowledge [15, 44], acquired by LLMs through exposure to the large scale internet corpus during training [1]. Prior studies have shown the ability of LLMs in knowledge internalization and generation[1, 40, 54, 60, 73], which knowledge is implicitly stored within model's parameters and retrieved during the generation process.

[*]We participated in Meta CRAG KDD Cup 2024 as Team ElectricSheep, securing third place in Task 1 and achieving first place in five of the seven question types in Task 2 among over 2, 000 participants and 5, 500 submissions. For access to the competition details and the leaderboard, please refer to the following URLs: https://www.aicrowd.com/challenges/meta-comprehensive-rag-benchmark-kdd-cup-2024 and https://discourse.aicrowd.com/t/meta-crag-challenge-2024-winners-announcement/10786.
[†]Ye Yuan is the leader of the team ElectricSheep.
[‡]The project was conducted as part of Gongbo Sun's research internship at the State Key Laboratory of Multimedia Information Processing, Peking University.
[§]Ming Zhang is the advisor of the team.

These models typically undergo a two-stage training process: pre-training and post-training. Pre-training uses large-scale, task-agnostic data, and often needs trillions of tokens [9, 24, 60, 69], with the objective of next token prediction. However, models emerging from pre-training are not aligned with human values and can produce harmful, biased, or toxic content [40]. To mitigate these issues, a post-training process is necessary, where techniques like supervised fine-tuning (SFT), Proximal Policy Optimization (PPO) [52] and Direct Preference Optimization (DPO) [47] are applied [9, 69].

Despite these advancements, under the paradigm of pre-training and fine-tuning, LLMs still face three critical challenges [19, 66, 70]:

1) Lack of domain expertise knowledge: LLMs may struggle with specialized domains like laws and medicine due to limited exposure during pre-training. Internal knowledge along learned as parameters is insufficient to comprehensively address complex legal or medical issues [34]. Fine-tuning the model's weights in these areas requires external computational resources, which is expensive and time-consuming [34, 40].

2) Hallucination during generation: Model can generate factually incorrect or inconsistent information [60]. For instance, when asked "Which number is larger, 3.11 or 3.9?", most LLMs, including GPT-4, incorrectly respond with 3.11 < 3.9. The hallucination may accumulate progressively when the model incorporates prompting techniques like Chain of Thought (CoT) [65], a method used to augment LLMs reasoning ability [61].

3) Difficulty integrating time-sensitive information: LLMs often lack up-to-date information since the knowledge stored within the model parameters is static and does not undergo synchronous updates over time [61], limiting their application in rapidly changing fields like sports and finance. These fields usually require real-time data processing such as querying the current price of a specific stock or the scores for a table tennis player at the Paris 2024 Olympics Games. However, previous research [61] indicates that LLMs like GPT-4 demonstrate an accuracy of less than 15% when answering both slow and fast-changing questions.

To address these issues, the Retrieval-Augmented Generation (RAG) approach has been introduced as a non-weight update approach that leverages external knowledge bases. A basic RAG system consists of two parts: a retriever and a generator. Relevant text documents are extracted from external knowledge sources and used as conditioning inputs alongside the query during generation [20]. The retriever accurately computes the similarity between user queries and external factual knowledge using metrics such as cosine similarity. The top most relevant sentences are extracted from external databases and combined with the inputs for the generator. This process enables a general LLM to acquire domain-specific knowledge from a corresponding domain database without sacrificing its generalization capabilities. Additionally, by combining retrieved facts with input queries, the hallucination problem is mitigated, leading to more accurate and informed responses. Furthermore, by maintaining an up-to-date database, time-dependent information can be seamlessly integrated into the LLMs. Consequently, RAG provides an effective approach for enhancing the capability of LLMs to generate domain-specific, factual, and time-sensitive responses.

In this paper, we introduce a novel Retrieval-Augmented Generation (RAG) system designed for real-world applications. Our system was evaluated on the CRAG benchmark [70], achieving the third position in Task 1 and securing first place for 5 out of 7 question types in Task 2. The rest of the paper is structured as follows. In Section 2, we provide an overview of related work in the context of RAG system design. We introduce the CRAG benchmark in Section 3. Our RAG system design is detailed in Section 4 and the complete system architecture is illustrated in Figure 1. The performance of our system is reported and discussed in Section 5. Finally, we conclude the paper in Section 6, including a Discussion Section of potential improvements for future iterations of our system.

2 Related Works

Numerous techniques have been proposed to address the aforementioned issues. For example, formal verification can help to reduce hallucination and output the verified reasoning process[32, 62, 67]. Moreover, there are some efficient training methods[41, 42, 53] that can help the model adapt to domain-specific knowledge. While numerous approaches have been proposed in the literature, the majority are tailored to address specific issues within a limited range of scenarios, making them unsuitable for direct application to CRAG tasks. We present a compilation of recent research to highlight the diverse design choices within the broader RAG research community. Drawing inspiration from recent advancements in the field, we have developed a novel design that integrates multiple strategies.

We built upon previous research [12, 19, 66, 73] to structure the conventional RAG workflow into four phases: pre-retrieval, retrieval, post-retrieval, and generation.

2.1 Pre-retrieval

The pre-retrieval phase involves indexing, query manipulation, and data modification. That is, compiling and indexing external knowledge sources to enhance the efficiency of subsequent searching execution, refining the query to better match the external data distribution and modifying data sources to create a coherent representation that encapsulates relevant knowledge for further reasoning.

Indexing: Text indexing has emerged as a prominent area of research within the field of data mining, leading to the development of numerous methodologies. Traditional indexing methods include the inverted index [74], suffix tree [37], and term frequency-inverse document frequency (TF-IDF) index [56]. These indexing methods utilize term frequency so that they may neglect the semantic information inherent in the text. Recent indexing methods employ language models such as BERT [7] and Sentence-T5 [39] to encode text into vectors within latent space. These vectors are used to construct vector indexes [8] that enhance similarity search through techniques including product quantization (PQ) [23], Hierarchical Navigable Small World (HNSW) [36], and Navigating Spreading-out Graph (NSG) [11].

Query Manipulation: The manipulation algorithms discussed below contribute to developing an improved metric for evaluating the similarity between user queries and external data. It is desirable to formulate a fully specified, context-independent query that effectively articulates user intent and aligns with the distribution of external knowledge sources. Early research primarily utilizes an n-gram language model derived from hundreds of millions of web search sessions [22] or employs a pointer-generator network to

improve referring expression resolution. Recent studies have leveraged the few-shot or zero-shot learning capabilities of language models that utilize a small set of manually created rewriting labels [71, 72]. In contrast to traditional query rewriting, which follows a "query in, query out" format, recent query manipulation approaches utilizing LLMs allow for "query in, passage out" approaches [35, 72].
Data Modification: Data extracted from original external knowledge sources may contain elements that are not conducive to the reasoning process of LLMs, such as redundant HTML tags found in web pages. However, these processes are closely linked to the format and distribution of external knowledge sources and often involve tedious text normalization procedures that are largely unrelated to the reasoning process, including tokenization, stemming, and the removal of stop words. Prior research has examined the use of substrings as document identifiers to improve search capabilities [4], as well as the application of LLMs to recite one or more relevant passages from their internal memory [58].

2.2 Retrieval

The retrieval phase primarily involves searching for pertinent materials from external knowledge sources and ranking them according to their relevance to the user query.
Search: Searching for relevant documents efficiently from vast external knowledge sources, such as the entire Internet, can be exceedingly challenging. Previous research on RAG that aims to retrieve local data typically employs local vector databases, such as Elasticsearch [16], or utilizes libraries like FAISS [8] for efficient similarity search and vector clustering. On the other hand, research focused on general-purpose RAG typically utilizes external web search engines directly; for instance, WebGPT [38] employs the Bing API.
Ranking: The ranking metric establishes the priority of each document housed in external knowledge sources. The metric used to evaluate the similarity between a potentially manipulated query and a document can be classified into two categories: sparse retrieval and dense retrieval. Sparse retrieval metrics, such as BM25 [20, 49] and Term Frequency–Inverse Document Frequency (TF-IDF) [5], rely on word frequency, whereas dense retrieval metrics utilize embedded vector, including Euclidean distance [29], cosine similarity [26], and Dense Passage Retrieval (DPR) [20, 28].

2.3 Post-retrieval

The post-retrieval phase includes re-ranking and filtering, which further refines the ranking results, and filtering out materials that are irrelevant to the querying topic.
Re-ranking: Although vector similarity-based document retrieval can be executed efficiently and effectively, including in parallel [25, 30] or distributed systems [10], its capacity to reveal semantic relationships between queries and documents remains constrained [57]. This is where re-ranking becomes significant: typically bigger models, which offer greater accuracy but lower efficiency, can precisely reorder the limited set of documents retrieved. Some approaches [14, 48] embed the query-document pair in a single pass, facilitating cross-attention between sequences. These approaches enhance the evaluation of mutual information between the texts. Other studies employ large language models as few-shot annotators to generate data for training a cross-attention re-ranker [6], or investigate the auto-regressive generation of re-ranking results to leverage inter-document relationships [18].
Filtering: Filtering seeks to remove redundant parts from the retrieval results, which may either be of poor quality or exhibit low relevance to the user query [64]. Current studies employ large language models to evaluate the utility of retrieved information and to critique whether all verification-worthy statements are substantiated by the associated documents [3, 63], or condense retrieved documents into a textual summary, thereby minimizing inference costs [68].

2.4 Generation

The curated reference materials are subsequently processed in the generation phase to produce the final results. Additionally, various enhancement techniques and customizations can be applied to the same phrase.
Enhancing: Enhancing, also known as retrieval fusion, seeks to improve the performance of large language model generations by utilizing retrieved documents. Typically, the retrieved documents are concatenated with the query in the expectation that the language models will generate coherent results based on the reference materials [17, 31, 48]. Other methods utilize sophisticated prompt templates that integrate multiple pieces of information [61] or employ techniques such as context compression, summarization, and filtering to achieve more efficient inference [2, 33, 64, 68]. Several studies have also investigated the integration of encoded retrieval vectors into input features [20, 21, 55].
Customization: During the customization process, it is crucial to meticulously refine the model's outputs to ensure alignment between the model's responses and the user's intended queries. This refinement will result in final answers that are concise, informative, and well-formatted. These approaches include generating reflection tokens for self-evaluation of the model's output [3] and implementing a graph-text contrastive learning method to enhance the alignment between generated content and retrieved references[27].

3 CRAG Benchmark

CRAG benchmark [70] is a factual question-answering benchmark with thousands of QA pairs and 50 real-world web pages for each data. The benchmark also provides a mock API for Knowledge Graph(KG) searching. The benchmark is set for the KDD Cup 2024 with $2,706$ data items available in public, half of which are for public validation. There are 5 domains and 8 question types in the benchmark, and each data item has a `static_or_dynamic` label that indicates whether the answer to a question changes and the expected rate of change, which can be used to analyze the model strengths and weaknesses. In order to mimic real-world application scenarios, each generated response is limited to 30 seconds on an AWS G4dn.12xlarge instance, which is equipped with 4 NVIDIA T4 GPUs providing a total of 64 GB of GPU memory during inference.

The benchmark is split into three tasks in the competition: Retrieval Summarization, Knowledge Graph and Web Retrieval, and End-to-End Retrieval Augmented Generation, which are from simple to complex. We introduce the three tasks as follows:

- **Task 1: Retrieval Summarization.** In this task, each question is provided with 5 web pages potentially containing relevant information to mimic the top 5 results from a real-world web search. They are created by storing up to 50 pages from search queries related to each question. Web pages are quite long, containing about $120,000$ tokens on average, to measure the CRAG systems' capability to identify and condense this information into accurate answers.
- **Task 2: Knowledge Graph and Web Retrieval.** Besides the web pages provided with each question, this task introduces mock APIs to access information from underlying mock Knowledge Graphs (KGs), with structured data possibly related to the questions. Mock KGs are created using the data behind the questions, supplemented with "hard negative" data to simulate a more challenging retrieval environment. Mock APIs facilitate structured searches within these KGs and can be used with parameters derived from the questions, to retrieve relevant data for answer formulation. The evaluation focuses on the systems' ability to query structured data and integrate information from various sources into comprehensive answers.
- **Task 3: End-to-End Retrieval Augmented Generation.** The third task increases complexity by providing 50 web pages and mock API access for each question, encountering both relevant information and noises. It assesses the systems' skill in selecting the most important data from a larger set, reflecting the challenges of real-world information retrieval and integration.

Each task builds upon the previous, employing more demanding tasks to gauge the CRAG system's capabilities in real-world and intricate scenarios.

As for evaluation, CRAG Benchmark employs both automated (Auto-eval) and human (Human-eval) evaluations. Auto-eval employs rule-based matching and GPT-4 assessment to check answer correctness. To promote concise answers, Auto-evaluators will only consider responses within 50 tokens. Human annotators will only decide the rating of top teams to reduce workload. In addition to correctness, human eval requires basic fluency for an answer to be considered Perfect.

Notably, the competition evaluates the system with a positive score for the correct answer, a zero score for answering "I don't know", and a negative score for the wrong answer, which is noted as `hallucination`. The benchmark not only encourages the model to answer the question correctly but also punishes the model for incorrect responses, which is closer to the real-world scenario.

4 System Design

The complete design of our system is shown in Figure 1. There are 6 critical modules in our system, including (1) web page processing, (2) attribute predictor, (3) numerical calculator, (4) LLM knowledge extractor, (5) KG Module, and (6) reasoning module. We have enhanced the system's capabilities in information extraction, reducing hallucinations, numerical calculation accuracy, and higher-order reasoning through these modules. Additionally, we have implemented special handling for corner cases. We will introduce these modules as follows.

4.1 Web Page Processing

Web pages serve as a shared information source for all three tasks, containing a substantial amount of potentially valuable information that can aid the model in task completion. As a result, web page processing is a critical component of system design, directly impacting both the quality of the extracted information and the accuracy of subsequent language model generations.

However, despite the abundance of information presented in natural language on web pages, extracting this information is not straightforward. This complexity is due to the frequent presence of significant amounts of noise that does not contribute relevant information necessary for task completion. Such noise can unnecessarily prolong the model's processing and reasoning time, potentially leading to misinterpretations. The types of noise encountered include decorative HTML tags used for typography, JavaScript code, and internal comments within the web page. While some HTML tags may contain semantic information that aids in paragraph segmentation or title identification, the useful information is not easy to extract. Moreover, there is some structured information in the HTML like tables, which will do harm to the text quality if they are improperly handled through methods such as truncation or splicing other texts. So we process the raw web page into two parts: text chunks and tables, to get the references with higher quality.

Text Chunks Processing. To address the challenges posed by the complexity of modern HTML web pages, we adopt `trafilatura`, a Python library specifically designed for gathering text from the Web. This library effectively mitigates noise generated by recurring elements such as headers, footers, and links. We also clean all the tables by identifying the `<table>` tag, which will be processed specially. To enhance robustness, we employ the classic `BeautifulSoup` library as a fallback option for a small subset of web pages that `trafilatura` cannot process. After extracting text using these two tools, we utilize functions provided by the `Blingfire` library to segment the text into individual sentences.

The meaning of a sentence is often influenced by its contextual placement; therefore, the information within a single sentence is often incomplete. We organize sentences into chunks based on the following rules to enhance semantic coherence in subsequent retrieval processes. First, we truncate individual sentences that exceed a predetermined length threshold to ensure they remain within an appropriate length. Next, we implement a keyword-based approach to identify whether a sentence is a question, utilizing indicators such as the 5W1H interrogatives at the beginning and the presence of question marks at the end. All keywords utilized in this process can be found in Appendix A. Sample data analysis has shown that questions are often immediately followed by their corresponding answers. Therefore, we concatenate questions with their subsequent text until we reach the pre-assigned length threshold. Finally, we connect any remaining ungrouped sentences in sets of 3.

Tables Processing. Given that the extracted text data typically does not include tables, we have employed `BeautifulSoup` to extract tables from web pages and convert them into Markdown format. We hypothesize that exposure to numerous documents formatted in Markdown during the model training phase will improve the model's understanding and interpretation of this format. Finally,

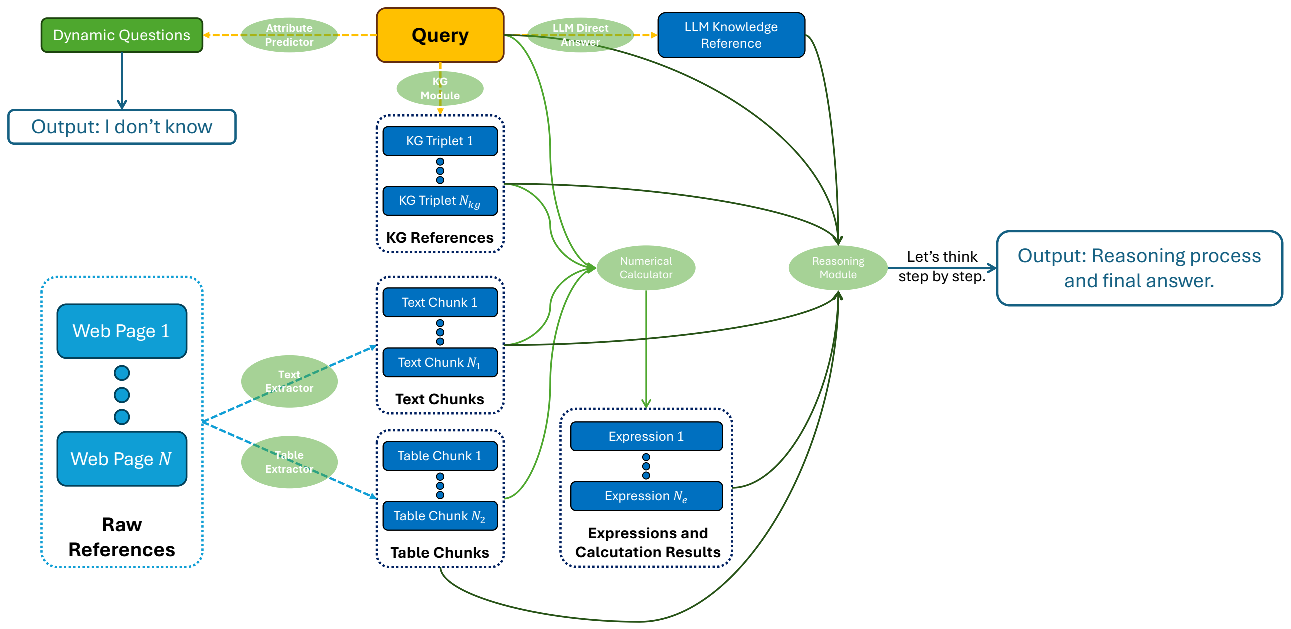

Figure 1: The complete design of our system. There are two possible routes for the generation. If the query is classified by the in-context learning as "dynamic", we will output "I don't know" directly to reduce hallucination on these hard problems.

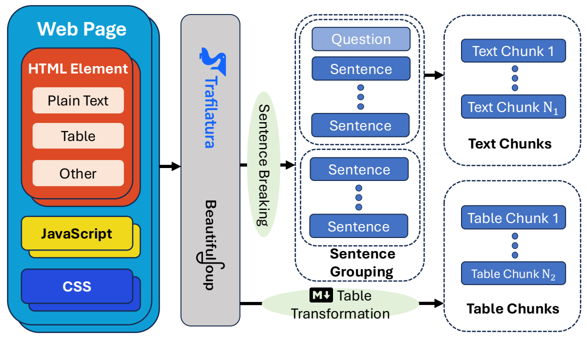

Figure 2: The design of our web page processing. We utilized Trafilatura and BeautifulSoup to extract plain text and tables from web pages. Following this extraction, we employed Blingfire to segment the plain text into sentences, which were then grouped into chunks based on heuristic methods. Additionally, the tables were converted into Markdown format for further processing.

we cleaned the empty tables to reduce the noise. The source code for table transformation is listed in Appendix B.

Text Embedding & Ranking Metrics. Regarding the ranking metrics and methodologies for retrieval, it is important to note that in the CRAG tasks, each question is associated with a relatively small number of candidate articles (five for each question in the initial two tasks). Consequently, we have adopted a straightforward approach. We utilized the `sentence-t5-large` model [39] to generate vector embeddings for both the text chunks and the queries, and we employed cosine similarity as the metric to rank the relevance of the text chunks. The cosine similarity between the user query embedding and a text chunk embedding $\mathbf{q}$ and $\mathbf{c}$ is defined as the cosine of the angle between them. Mathematically, it is given by:

$$\text{cosine\_similarity} = \cos(\theta) = \frac{\mathbf{q} \cdot \mathbf{c}}{\|\mathbf{q}\| \|\mathbf{c}\|}$$

where $\mathbf{q} \cdot \mathbf{c}$ is the dot product of embedding vectors $\mathbf{q}$ and $\mathbf{c}$, and $\|\mathbf{q}\|$ and $\|\mathbf{c}\|$ are their respective magnitudes. Our experiments indicated that employing LLMs for complex query manipulation methods did not significantly improve retrieval accuracy; instead, it resulted in substantial computational overhead. Therefore, we primarily leveraged the LLM knowledge extractor, as detailed below, to enhance the quality of the references.

4.2 Attribute Predictor

Large language models demonstrate considerable variability in their performance across various question-answering tasks. Within the context of the CRAG tasks, answering aggregation questions and multi-hop questions poses greater challenges than addressing simple questions. This difficulty arises from the model's need to not only possess robust information retrieval capabilities but also to integrate multiple sources of information and engage in reasoning processes. Moreover, when dealing with questions related to slow-changing and fast-changing facts, the model must exhibit temporal awareness, which adds complexity to the generation of accurate responses [61]. To tackle these challenges, we have developed an attribute predictor that assesses the type of each specific question and the rate of underlying factual change, aiming to optimize performance across all question types.

There are three useful attributes in the benchmark: `domain`, `question_type`, and `static_or_dynamic`. Following the descriptions outlined in the CRAG dataset, we classified the `domain` into five categories: Finance, Sports, Music, Movies, and Encyclopedia Open Domain. The `question_type` was divided into the following categories: simple question, simple question with some condition, set question, comparison question, aggregation question, multi-hop question, post-processing question, and false premise question. However, the `question_type` attribute is hard to predict because it often needs reasoning through all references, so we didn't apply attribute prediction to it. Additionally, the `static_or_dynamic` attribute, which pertains to timeliness, was classified into four categories: real-time, fast-changing, slow-changing, and stable. Moreover, we found that the boundaries between these four categories are not clear, so we only classified this attribute into two categories: static and dynamic. For each attribute, we implemented two methods for question classification: one leveraging the in-context learning capability of large language models and the other employing support vector machines (SVM).

In-Context Learning. Large language models demonstrate robust natural language understanding and strong multi-task generalization abilities. We prompt the model with classification instructions and 5 demonstrations for categories, instructing it to classify subsequent questions. The demonstrations are randomly selected from the publicly available CRAG dataset. All prompts utilized in the classification process can be found in Appendix C.1.

To enhance classification reliability, we adopted a self-consistency strategy involving multiple samplings from the large language model. The category that appeared most frequently among the sampled results was designated as the classification for the question.

SVM. We also tried to train an SVM classifier using the CRAG public dataset to reduce computational overhead. We used the `all-MiniLM-L6-v2` model to get the sentence embeddings, which are used to train the SVM. We observed that the SVM model can get higher accuracy in predicting the attributes with less time and computation consumption. However, we didn't have time to merge the code into our final version for evaluation. So, consequently, the submitted version relied on the few-shot learning approach with the large language model for classification, and we leave the improvement for this module as future work. We will show the detailed results analysis and comparison in Section 5.1.

4.3 Numerical Calculator

Previous research has highlighted the phenomenon of "hallucination" in large language models, particularly concerning their performance in precise numerical calculations [13, 46]. Within the framework of the CRAG tasks, answers to aggregation questions, multi-hop questions, and post-processing questions are not directly available in the retrieved content. Instead, these tasks require the model to derive the final answer through a series of reasoning steps, which often involve precise numerical computations, especially in the domain of finance and sports. This requirement poses additional challenges to the model's capacity to generate accurate results.

To address this issue, we employed an approach that leverages external tools, drawing inspiration from previous research [43, 51, 73]. We encourage the large language model to articulate the reasoning steps necessary to solve the problem as mathematical expressions while delegating the actual numerical calculations to an external Python interpreter. We specifically integrate retrieved text chunks and tables that may contain numerical information into the model's prompts and employ prompt techniques that encourage the model to generate valid Python expressions directly. Detailed specifications of the prompts are provided in the Appendix C.2 In the submitted version, we utilized multiple sampling and processed the generated Python expressions using the `eval` function.

Note that the program code generated by the LLM may include malicious code, and executing such code directly poses a potential threat to system stability. To mitigate such risk, the best practice for ensuring system security is to use `ast.literal_eval` or to execute the code within a sandbox environment. We leave the safety and proper termination of sandbox execution as future work.

4.4 LLM Knowledge Extractor

Through extensive training on diverse corpora, LLMs have acquired substantial knowledge and demonstrated robust question-answering capabilities across a wide range of domains. Chances are that the reference documents are not beneficial in generating accurate responses to the model's queries. This may be attributed to the possibility that the retrieved materials are outdated, or they may include irrelevant or even misleading information. Consequently, these reference documents do not enhance the model's capacity to produce answers but harm it. Furthermore, during the generation phase, we employ the Llama 3 instruction-tuned models that are optimized for dialogue use cases and possess a robust capability for following instructions. Our findings indicate that when reference documents are provided within the prompt, the model exhibits a significant tendency to extract answers from these documents, regardless of whether the documents contain the necessary information. This inclination may be a result of the instructional fine-tuning process. In contrast, LLMs that operate without any references are capable of accurately answering these questions. In light of this observation and drawing inspiration from prior research [58], we developed a large language model knowledge extractor. This extractor leverages the knowledge-rich responses generated by the large language model as part of the reference materials for enhanced reasoning.

The process of extracting knowledge from the model closely resembles the normal model generation process. It similarly utilizes zero-shot indications, which include prompts requiring the model to assess whether a given query pertains to a false-premise issue and to generate more concise responses. However, a notable distinction exists in the lack of reference documents sourced from external knowledge bases within the prompts, as well as the exclusion of multiple sampling, which is intended to reduce computational overhead. In this way, the LLM could respond solely based on the knowledge internalized within its parameters during the training. Our findings suggest that this approach results in favorable performance on questions classified as slow-changing and stable. We anticipate that this approach will effectively align the knowledge embedded in the LLM's parameters with external reference documents, thereby mitigating the issue of the model's excessive dependence on externally retrieved information. Moreover, we also use the zero-shot CoT to let the model reasoning by itself for more accurate knowledge. The prompt template is shown in Appendix C.3

However, as described previously, letting the model directly answer the questions will introduce hallucinations in its knowledge, although with zero-shot CoT reasoning. To balance the hallucination and knowledge from the LLM itself, we only treat the output of this module as one of the references. We have carefully designed prompts to ensure that the model neither overly relies on document references nor excessively trusts the LLM's knowledge. Section 4.6 will provide a more detailed introduction.

4.5 Knowledge Graph Module

In addition to web references, Task 2 and Task 3 also provide a mock API for querying the provided knowledge graph (KG). As a structured knowledge base, a KG provides accurate information. However, the generation of a KG query is crucial to determining whether the system can retrieve the correct answer. We started from the KG baseline, which extracted the entities in the query with an LLM, and generated the query by manual rules. The quality of rule-based queries is limited by the complexity of the rules, and hard to scale. So we tried a function-calling method, which makes all the mock APIs as the input of the LLM, and lets it generate a proper function calling. However, due to limitations in time and resources, we were unable to optimize the models and prompts for the function-calling method, resulting in suboptimal performance. Therefore, we reverted to the KG baseline method and did not make further improvements in the submitted version. We show the prompts for the function-calling methods in Appendix C.4.

4.6 Reasoning Module

After all the previously introduced processing methods, we get text chunks, tables, triplets from KG, and knowledge from LLM weights as the references. We carefully designed a prompt template to let the LLM do reasoning from all these references and get the final answer. We control the reasoning process by output format demonstration and zero-shot CoT, which is useful for multi-hop questions. Leveraging the strong instruction-following capabilities of Llama3-70B-Instruct, we've successfully maintained steady progress in controlling reasoning tasks. We designed several rules to constrain the reasoning path and output format, including that the output should be precise, and guide the model reasoning by asking intermediate questions in the prompt. The complete prompt is shown in Appendix C.5.

4.7 Handling Corner Cases

In addition to the main modules mentioned above, we have also handled many corner cases, including (1) identifying invalid questions; (2) encouraging the model to answer "I don't know" for unsure answers to reduce hallucination; and (3) dealing with outputs that do not comply with the instruction format. We will introduce our design to handle these corner cases as follows.

Invalid Questions. There are some questions that have false premises, which means the query is contradictory to the fact. For these questions, the model should output "invalid questions". To identify this type of question, the model needs to carefully analyze the references provided. We add special rules in the reasoning prompt shown in Appendix C.5

	Correct(%)↑	Missing(%)	Hallucination(%)↓	Score(%)↑
Official LLM Baseline	16.2	0.0	83.7	-67.6
Official RAG Baseline	25.4	2.5	72.1	-46.6
Our System	**29.7**	**56.3**	**13.9**	**15.8**

Table 1: The main results of our designed system evaluated in the public test dataset.

Reduce Hallucination. We employed two approaches to alleviate hallucinations: attribute prediction and reasoning. We found that the time-changing questions, which would be labeled as `dynamic` by attribute predictor, are hard for our system and we do not have enough time and resources to improve them. So we manually let the system answer "I don't know" for these questions. Moreover, we added several rules and prompt engineering techniques in the reasoning module to let the model answer "I don't know" when it is unsure. Ultimately, we configured the system to exclusively output "I don't know" and refrain from adding any additional words whenever "I don't know" is included in the initial response.

Incorrect Format. Cause we didn't conduct constrained sampling for the reasoning output, there is the possibility that the model will output answers that can not be parsed. To handle this situation, we design a backup summarization agent to summarize the final answer precisely and concisely based on the reasoning module's output when the parse fails. The prompt for this module is shown in Appendix C.6

5 Experiments

We conducted ample experiments to verify the effectiveness of each module. The main results of our local evaluation are shown in Table 1, where we got a lot of improvement in the public test set compared with the baseline of Task 1. We got a 15.8% score where we greatly reduced the hallucination ratio and changed these hallucinations into answering "I don't know". We show detailed analysis and ablation studies of our evaluation results. In the final private test, we got a 21.8% score in Task 1. We will also show an analysis of the private evaluation.

5.1 Detailed Analysis

To gain a deeper understanding of the strengths and weaknesses of our system across various aspects, we conducted a meticulous analysis of the evaluation results on the public test set. Figure 3 shows the detailed scores in Task 1 setting.

For the domain attribute, we perform well in areas such as movies, music, and open topics, but our performance is lacking in finance and sports. This is due to the fact that these two domains require the model to have the ability to answer dynamic information that changes over time. To prevent hallucinations, our model opts to refuse to answer such queries. The performance regarding the attribute of dynamism reaches the same conclusion: as the dynamism of the model increases, the effectiveness of our system gradually declines. Regarding the score distribution across question types, our analysis indicates that the system exhibits superior performance on tasks requiring complex reasoning, which benefits from the robust functionality of our integrated reasoning module.

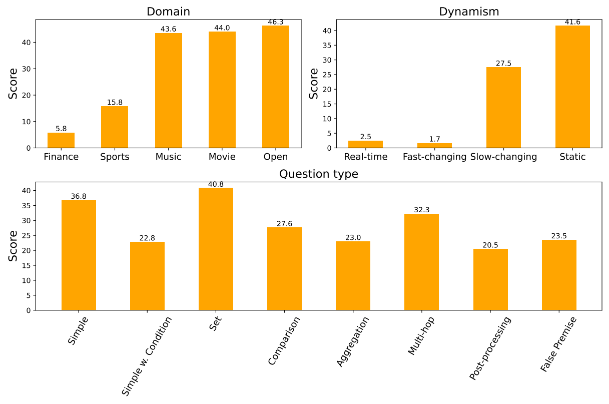

Figure 3: Detailed score across different attributes in the local evaluation of Task 1.

	Correct (%)↑	Missing (%)	Hallucination (%)↓	Score↑
Official RAG Baseline	16.2	0.0	83.8	-67.6
+ Optimized Chunk Extraction	23.7	0.0	76.3	-52.6
+ Llama3-70B-GPTQ + Prompt Refinement	24.4	50.8	24.8	-0.3
+ Attribute Predictor	19.9	65.4	14.7	5.2
+ Table Extractor	21.3	63.4	15.2	6.1
+ Numerical Calculator	23.7	29.7	16.6	7.2
+ Reasoning Module	20.9	67.3	11.8	9.1
+ False Premise Identify	26.0	60.5	13.5	12.5
+ LLM Knowledge Extractor	27.3	59.3	13.4	13.9
+ KG Module (Final System)	**29.7**	**56.3**	**13.9**	**15.8**

Table 2: Ablation results of our system. The modules are added to the system gradually.

5.2 Ablation Study

We performed extensive experiments to validate the enhancements contributed by individual components of our system. We developed the system incrementally, systematically verifying and integrating beneficial modules. Consequently, comprehensive ablation studies, which would involve removing each component from the final system, were not conducted. Instead, we documented the rationale behind the inclusion of each module and its resulting improvements. Table 2 outlines the primary construction pathway of our system.

We started from the baseline model and did a lot of refinement and added modules to it. As shown in Table 2, each module we added would increase the final score. The main optimization directions are reducing hallucinations and increasing correctness.

5.3 Analysis For Private Evaluation

Although the competition organizers have not provided a comprehensive and detailed analysis of the results on the private leaderboard, we can still present the currently published results and analyze our system. Table 3 shows the score for Task 1 in private evaluation. Our system achieved scores close to the champion in Task 1 but fell significantly behind in Task 2 and Task 3. We believe this is due to our underutilization of the information from the knowledge graph. Table 4 shows the prizes we won from 5 out of 7 question types in Task 2. We find that our system performs well on question types that require complex reasoning, such as aggregation and multi-hop, which we attribute to our reasoning module.

	Task 1			Task 2			Task 3		
Team	db3	md_dh	**ElectricSheep**	db3	APEX	md_dh	db3	APEX	vslyu-team
Score(%)	28.4	24.0	21.8	42.7	41.0	31.0	47.8	44.9	25.6

Table 3: The evaluation results in the private test set.

Furthermore, Figure 4 illustrates the scores across various attributes, revealing that our focus has been on static and slow-changing questions, neglecting the challenging time-varying ones. This deficiency in handling dynamic questions has also led to suboptimal performance in financial question types. The score results align with those from our internal assessment; however, we encountered a higher number of incorrect responses in the movie domain and on simple question types. We hypothesize that this discrepancy may stem from variations in question distribution between our local evaluation and online test datasets. Additionally, our performance in popularity aligns well with the expectations set forth in the CRAG Benchmark [70].

Question Type	Team	Score(%)
simple_w_condition	**ElectricSheep**	23.9
set	**ElectricSheep**	36.65
comparison	dRAGonRAnGers	38
aggregation	**ElectricSheep**	18.75
multi_hop	**ElectricSheep**	23.2
post_processing	**ElectricSheep**	11.75
false_premise	Future	64.6

Table 4: The scores of the prizes we won in Task 2.

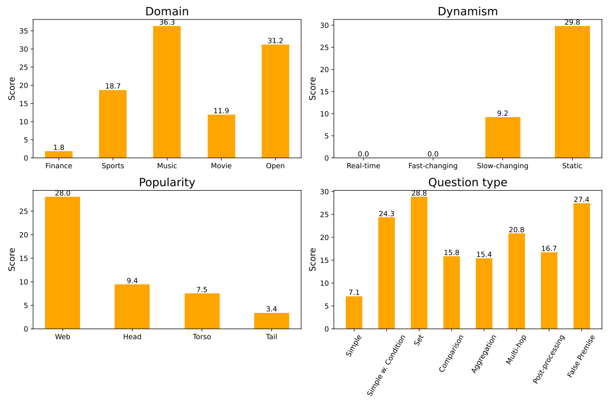

Figure 4: Detailed score across different attributes in the online private evaluation Task 1.

6 Conclusion

With the designed RAG system, we finally got 3rd place in task 1 and got the prize for 5 out of 7 question types in task 2. The final evaluation scores can be found in the Winners Announcement. The table extractor, reasoning module, and calculator module have demonstrated substantial enhancements over the baseline system. Additionally, the ICL attribute predictor has significantly reduced hallucinations in responses to difficult questions.

Discussions

Many aspects of our system can be improved. For example, we only used two tower models for retrieval, while re-ranker models are more suitable for task 1. Moreover, a two-stage retrieval and re-ranker system should be used for task 3. We didn't optimize the KG information retrieval in task 2, which can be improved a lot in the future. The current methods for handling tables are relatively simple. Some tables are useless or too large with a lot of noise, but we didn't handle these cases. There should be retrieval and structural query methods specifically designed for tables.

Acknowledgments

This project is partially supported by the National Key Research and Development Program of China with Grant No. 2023YFC3341203 as well as the National Natural Science Foundation of China with Grant Number 62276002.

We thank Prof. Ming Zhang's mentorship and support to the team and Alan Wu's assistance and discussions.

References

[1] Badr AlKhamissi, Millicent Li, Asli Celikyilmaz, Mona Diab, and Marjan Ghazvininejad. 2022. A Review on Language Models as Knowledge Bases. arXiv:cs.CL/2204.06031 https://arxiv.org/abs/2204.06031
[2] Md Adnan Arefeen, Biplob Debnath, and Srimat Chakradhar. 2024. Leancontext: Cost-efficient domain-specific question answering using llms. *Natural Language Processing Journal* 7 (2024), 100065.
[3] Akari Asai, Zeqiu Wu, Yizhong Wang, Avirup Sil, and Hannaneh Hajishirzi. 2023. Self-rag: Learning to retrieve, generate, and critique through self-reflection. *arXiv preprint arXiv:2310.11511* (2023).
[4] Michele Bevilacqua, Giuseppe Ottaviano, Patrick Lewis, Scott Yih, Sebastian Riedel, et al. 2022. Autoregressive search engines: Generating substrings as document identifiers. *Advances in Neural Information Processing Systems* 35 (2022), 31668–31683.
[5] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051* (2017).
[6] Zhuyun Dai, Vincent Y Zhao, Ji Ma, Yi Luan, Jianmo Ni, et al. 2022. Promptagator: Few-shot dense retrieval from 8 examples. *arXiv preprint arXiv:2209.11755* (2022).
[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[8] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, et al. 2024. The Faiss library. (2024). arXiv:cs.LG/2401.08281
[9] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).
[10] Karima Echihabi, Kostas Zoumpatianos, and Themis Palpanas. 2021. New trends in high-d vector similarity search: al-driven, progressive, and distributed. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3198–3201.
[11] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. 2017. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143* (2017).
[12] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, et al. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).
[13] Mor Geva, Ankit Gupta, and Jonathan Berant. 2020. Injecting numerical reasoning skills into language models. *arXiv preprint arXiv:2004.04487* (2020).
[14] Michael Glass, Gaetano Rossiello, Md Faisal Mahbub Chowdhury, Ankita Rajaram Naik, Pengshan Cai, et al. 2022. Re2G: Retrieve, rerank, generate. *arXiv preprint arXiv:2207.06300* (2022).
[15] Yoav Goldberg. 2019. Assessing BERT's Syntactic Abilities. arXiv:cs.CL/1901.05287 https://arxiv.org/abs/1901.05287
[16] Clinton Gormley and Zachary Tong. 2015. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.".
[17] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Mingwei Chang. 2020. Retrieval augmented language model pre-training. In *International conference on machine learning*. PMLR, 3929–3938.
[18] Sebastian Hofstätter, Jiecao Chen, Karthik Raman, and Hamed Zamani. 2023. Fid-light: Efficient and effective retrieval-augmented text generation. In *Proceedings of the 46th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 1437–1447.
[19] Yizheng Huang and Jimmy Huang. 2024. A Survey on Retrieval-Augmented Text Generation for Large Language Models. *arXiv preprint arXiv:2404.10981* (2024).
[20] Gautier Izacard and Edouard Grave. 2021. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. arXiv:cs.CL/2007.01282 https://arxiv.org/abs/2007.01282
[21] Gautier Izacard, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, et al. 2023. Atlas: Few-shot learning with retrieval augmented language models. *Journal of Machine Learning Research* 24, 251 (2023), 1–43.
[22] Bernard J Jansen, Danielle L Booth, and Amanda Spink. 2009. Patterns of query reformulation during web searching. *Journal of the american society for information science and technology* 60, 7 (2009), 1358–1371.
[23] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2010. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2010), 117–128.
[24] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, et al. 2023. Mistral 7B. arXiv:cs.CL/2310.06825 https://arxiv.org/abs/2310.06825
[25] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[26] Kush Juvekar and Anupam Purwar. 2024. Cos-mix: cosine similarity and distance fusion for improved information retrieval. *arXiv preprint arXiv:2406.00638* (2024).
[27] Minki Kang, Jin Myung Kwak, Jinheon Baek, and Sung Ju Hwang. 2023. Knowledge graph-augmented language models for knowledge-grounded dialogue generation. *arXiv preprint arXiv:2305.18846* (2023).

[28] Vladimir Karpukhin, Barlas Oğuz, Sewon Min, Patrick Lewis, Ledell Wu, et al. 2020. Dense passage retrieval for open-domain question answering. *arXiv preprint arXiv:2004.04906* (2020).

[29] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172* (2019).

[30] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 39–48.

[31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[32] Chengwu Liu, Jianhao Shen, Huajian Xin, Zhengying Liu, Ye Yuan, et al. 2023. Fimo: A challenge formal dataset for automated theorem proving. *arXiv preprint arXiv:2309.04295* (2023).

[33] Junyi Liu, Liangzhi Li, Tong Xiang, Bowen Wang, and Yiming Qian. 2023. Tcra-llm: Token compression retrieval augmented large language model for inference cost reduction. *arXiv preprint arXiv:2310.15556* (2023).

[34] Cui Long, Yongbin Liu, Chunping Ouyang, and Ying Yu. 2024. Bailicai: A Domain-Optimized Retrieval-Augmented Generation Framework for Medical Applications. arXiv:cs.CL/2407.21055 https://arxiv.org/abs/2407.21055

[35] Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting for retrieval-augmented large language models. *arXiv preprint arXiv:2305.14283* (2023).

[36] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[37] Edward M McCreight. 1976. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)* 23, 2 (1976), 262–272.

[38] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332* (2021).

[39] Jianmo Ni, Gustavo Hernandez Abrego, Noah Constant, Ji Ma, Keith B Hall, et al. 2021. Sentence-t5: Scalable sentence encoders from pre-trained text-to-text models. *arXiv preprint arXiv:2108.08877* (2021).

[40] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, et al. 2022. Training language models to follow instructions with human feedback. arXiv:cs.CL/2203.02155 https://arxiv.org/abs/2203.02155

[41] Yu Pan, Ye Yuan, Yichun Yin, Jiaxin Shi, Zenglin Xu, et al. 2024. Preparing Lessons for Progressive Training on Language Models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 18860–18868.

[42] Yu Pan, Ye Yuan, Yichun Yin, Zenglin Xu, Lifeng Shang, et al. 2023. Reusing pretrained models by multi-linear operators for efficient training. *Advances in Neural Information Processing Systems* 36 (2023), 3248–3262.

[43] Bhargavi Paranjape, Scott Lundberg, Sameer Singh, Hannaneh Hajishirzi, Luke Zettlemoyer, et al. 2023. Art: Automatic multi-step reasoning and tool-use for large language models. *arXiv preprint arXiv:2303.09014* (2023).

[44] Matthew E. Peters, Mark Neumann, Luke Zettlemoyer, and Wen tau Yih. 2018. Dissecting Contextual Word Embeddings: Architecture and Representation. arXiv:cs.CL/1808.08949 https://arxiv.org/abs/1808.08949

[45] Fabio Petroni, Tim Rocktäschel, Patrick Lewis, Anton Bakhtin, Yuxiang Wu, et al. 2019. Language Models as Knowledge Bases? arXiv:cs.CL/1909.01066 https://arxiv.org/abs/1909.01066

[46] Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. *arXiv preprint arXiv:2009.03393* (2020).

[47] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D. Manning, et al. 2024. Direct Preference Optimization: Your Language Model is Secretly a Reward Model. arXiv:cs.LG/2305.18290 https://arxiv.org/abs/2305.18290

[48] Ori Ram, Yoav Levine, Itay Dalmedigos, Dor Muhlgay, Amnon Shashua, et al. 2023. In-context retrieval-augmented language models. *Transactions of the Association for Computational Linguistics* 11 (2023), 1316–1331.

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

[50] Tara Safavi and Danai Koutra. 2021. Relational World Knowledge Representation in Contextual Language Models: A Review. arXiv:cs.CL/2104.05837 https://arxiv.org/abs/2104.05837

[51] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, et al. 2023. Toolformer: language models can teach themselves to use tools. 2023. *arXiv preprint arXiv:2302.04761* (2023).

[52] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:cs.LG/1707.06347 https://arxiv.org/abs/1707.06347

[53] Jianhao Shen, Chenguang Wang, Ye Yuan, Jiawei Han, Heng Ji, et al. 2022. PALT: Parameter-lite transfer of language models for knowledge graph completion. *arXiv preprint arXiv:2210.13715* (2022).

[54] Jianhao Shen, Ye Yuan, Srbuhi Mirzoyan, Ming Zhang, and Chenguang Wang. 2024. Measuring vision-language stem skills of neural models. *arXiv preprint arXiv:2402.17205* (2024).

[55] Devendra Singh, Siva Reddy, Will Hamilton, Chris Dyer, and Dani Yogatama. 2021. End-to-end training of multi-document reader and retriever for open-domain question answering. *Advances in Neural Information Processing Systems* 34 (2021), 25968–25981.

[56] Karen Sparck Jones. 1972. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation* 28, 1 (1972), 11–21.

[57] Harald Steck, Chaitanya Ekanadham, and Nathan Kallus. 2024. Is cosine-similarity of embeddings really about similarity?. In *Companion Proceedings of the ACM on Web Conference 2024*. 887–890.

[58] Zhiqing Sun, Xuezhi Wang, Yi Tay, Yiming Yang, and Denny Zhou. [n. d.]. Recitation-Augmented Language Models. In *The Eleventh International Conference on Learning Representations*.

[59] Gemini Team, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, et al. 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:cs.CL/2312.11805 https://arxiv.org/abs/2312.11805

[60] OpenAI Team. 2024. GPT-4 Technical Report. arXiv:cs.CL/2303.08774 https://arxiv.org/abs/2303.08774

[61] Tu Vu, Mohit Iyyer, Xuezhi Wang, Noah Constant, Jerry Wei, et al. 2023. FreshLLMs: Refreshing Large Language Models with Search Engine Augmentation. arXiv:cs.CL/2310.03214 https://arxiv.org/abs/2310.03214

[62] Haiming Wang, Ye Yuan, Zhengying Liu, Jianhao Shen, Yichun Yin, et al. 2023. Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 12632–12646.

[63] Haoyu Wang, Tuo Zhao, and Jing Gao. 2024. BlendFilter: Advancing Retrieval-Augmented Large Language Models via Query Generation Blending and Knowledge Filtering. *arXiv preprint arXiv:2402.11129* (2024).

[64] Zhiruo Wang, Jun Araki, Zhengbao Jiang, Md Rizwan Parvez, and Graham Neubig. 2023. Learning to filter context for retrieval-augmented generation. *arXiv preprint arXiv:2311.08377* (2023).

[65] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, et al. 2023. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. arXiv:cs.CL/2201.11903 https://arxiv.org/abs/2201.11903

[66] Shangyu Wu, Ying Xiong, Yufei Cui, Haolun Wu, Can Chen, et al. 2024. Retrieval-Augmented Generation for Natural Language Processing: A Survey. arXiv:cs.CL/2407.13193 https://arxiv.org/abs/2407.13193

[67] Jing Xiong, Jianhao Shen, Ye Yuan, Haiming Wang, Yichun Yin, et al. 2023. Trigo: Benchmarking formal mathematical proof reduction for generative language models. *arXiv preprint arXiv:2310.10180* (2023).

[68] Fangyuan Xu, Weijia Shi, and Eunsol Choi. 2023. Recomp: Improving retrieval-augmented lms with compression and selective augmentation. *arXiv preprint arXiv:2310.04408* (2023).

[69] An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, et al. 2024. Qwen2 Technical Report. arXiv:cs.CL/2407.10671 https://arxiv.org/abs/2407.10671

[70] Xiao Yang, Kai Sun, Hao Xin, Yushi Sun, Nikita Bhalla, et al. 2024. CRAG – Comprehensive RAG Benchmark. arXiv:cs.CL/2406.04744 https://arxiv.org/abs/2406.04744

[71] Shi Yu, Jiahua Liu, Jingqin Yang, Chenyan Xiong, Paul Bennett, et al. 2020. Few-shot generative conversational query rewriting. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 1933–1936.

[72] Wenhao Yu, Dan Iter, Shuohang Wang, Yichong Xu, Mingxuan Ju, et al. 2022. Generate rather than retrieve: Large language models are strong context generators. *arXiv preprint arXiv:2209.10063* (2022).

[73] Ye Yuan, Kexin Tang, Jianhao Shen, Ming Zhang, and Chenguang Wang. 2024. Measuring Social Norms of Large Language Models. *arXiv preprint arXiv:2404.02491* (2024).

[74] Justin Zobel and Alistair Moffat. 2006. Inverted files for text search engines. *ACM computing surveys (CSUR)* 38, 2 (2006), 6–es.

A Question Identification

The source code we used to identify questions is shown as follows:

```python
START_WORDS = [
    "who", "what", "when", "where", "why",
    "how", "is", "can", "does", "do", "did",
    "will", "would", "could", "should",
    "are", "was", "were",
    "has", "have", "had",
    "which", "whom", "whose",
]
def is_question(sentence: str):
    return sentence.lower().endswith("?") or \
           sentence.lower().startswith(tuple(START_WORDS))
```

B Table Transformation

The code we used for table transformation is shown as follows.

```python
from bs4 import BeautifulSoup
def clean_html(webpage, remove_links=False):
    soup = BeautifulSoup(webpage, features="html.parser")
    for script in soup.find_all(["script", "style", "meta"]):
        script.decompose()
    if remove_links:
        for link in soup.find_all("a"):
            link.unwrap()
    for footer in soup.find_all("footer"):
        footer.decompose()
    for button in soup.find_all("button"):
        button.decompose()
    for text_element in soup.find_all(String=True):
        if "\\u" in text_element or "\\U" in text_element:
            # Decode the escaped unicode characters
            decoded_text = bytes(text_element, "utf-8").decode("unicode_escape")
            text_element.replace_with(decoded_text)
    return soup
def check_empty_table(table_str: str) -> bool:
    table_str = (
        table_str.replace("|", "")
        .replace("-", "")
        .replace(" ", "")
        .replace("\n", "")
        .strip()
    )
    table_str = "".join(table_str.split())
    if table_str == "":
        return True
    return False
def get_tables(soup, html_name: str) -> list[str]:
    markdown_tables = []
    # Find all table tags
    tables = soup.find_all("table")
    for table in tables:
        markdown_table = []
        rows = table.find_all("tr")
        for row in rows:
            cells = row.find_all(["th", "td"])
            # Extract text from each cell and join them with a pipe
            formatted_row = (
                "| " + " | ".join(cell.get_text(strip=True) for cell in cells) + " |"
            )
            markdown_table.append(formatted_row)
        # Add a separator after the header (assumes the first row is header)
        if markdown_table:
            header_sep = (
                "| "
                + " | ".join("---" for _ in markdown_table[0].split("|")[1:-1])
                + " |"
            )
            markdown_table.insert(1, header_sep)
        markdown_table_str = "\n".join(markdown_table)
        if check_empty_table(markdown_table_str):
            continue
        markdown_table_str = f"Page name: {html_name}\n{markdown_table_str}"
        markdown_tables.append(markdown_table_str)
    return markdown_tables
```

C Prompt Details

C.1 Prompt for Attribute Predictor

Here we show the prompt we used for the attribute predictor. Specifically, we show the prompt template for dynamic prediction.

```
{
"system_prompt": "You will be provided with a question. Your task is to identify whether this
        question is a static question or a dynamic question. A static question is that the
        answer is fixed and will not change over time. A dynamic question is that the answer
        will change over time or needs time information. You **MUST** choose from one of the
        following choices: [\"static\", \"dynamic\"]. You **MUST** give the question type
        succinctly, using the fewest words possible.\nHere are some examples:\n" + \
    "------\n### Question: {}\n### Static or Dynamic: {}\n\n".format(
        example["query"],
        example["static_or_dynamic"]
    ) * few_shot_num
"user_prompt": "Here is the question: {query}\nRemember your rule: You **MUST** choose from
        the following choices: [\"static\", \"dynamic\"].\nWhat is the static or dynamic of
        this question?".format(query=query)
}
```

C.2 Prompt for Numerical Calculator

Here we show the prompt template for the numerical calculator.

```python
class CalcAgent:
    def __init__(self, max_table_length=6000):
        self.max_table_length = max_table_length
        self.expression_sample_num = 5

    def format_expression_prompts(
        self,
        batch_queries: list[str],
        batch_retrieval_results: list[list[str]],
        batch_tables: list[list[str]],
        tokenizer: AutoTokenizer,
    ) -> list[str]:
        system_prompt = """You are provided with a question and various references. Your task
                is to generate a possible useful expression that is needed to answer the
                question. Here are the rules:
1. The expression **MUST** be a valid Python expression.
2. The expression **MUST** be useful to answer the question.
3. If you think no expression is needed, you **MUST** answer with empty string.
4. The output should be succinct, you **MUST** do reasoning in your heart without outputing
        the reasoning.
5. You **MUST NOT** output any other words except the valid Python expression.
6. You **MUST NOT** output the expression that need the user to input anything.
"""
        formatted_prompts = []
        for _idx, query in enumerate(batch_queries):
            retrieval_results = batch_retrieval_results[_idx]
            related_tables = batch_tables[_idx]
            user_message = ""
            references = ""
            if len(retrieval_results) > 0:
                references += "# References \n"
                # Format the top sentences as references in the model's prompt template.
                for _snippet_idx, snippet in enumerate(retrieval_results):
                    references += f"- {snippet.strip()}\n"
            user_message += f"{references}\n------\n\n"

            if len(related_tables) > 0:
                table_references = ""
                user_message += "## Table references \n"
                for idx, table in enumerate(related_tables):
                    table_references += f"### Table {idx + 1}: \n"
                    table_references += f"{table}\n"
                table_references = table_references[: self.max_table_length]
                user_message += f"{table_references}\n------\n\n"
            user_message += "**Remember your rules**:"
            user_message += """
1. The expression **MUST** be a valid Python expression.
2. The expression **MUST** be useful to answer the question.
3. If you think no expression is needed, you **MUST** answer with empty string.
4. The output should be succinct, you **MUST** do reasoning in your heart without outputing
        the reasoning.
5. You **MUST NOT** output any other words except the valid Python expression.
6. You **MUST NOT** output the expression that need the user to input anything.
"""
            user_message += f"Question: {query}\n"
            user_message += f"Using the references listed above and based on the question,
                generate a valid Python expression for me: \n"

            formatted_prompts.append(
                tokenizer.apply_chat_template(
                    [
                        {"role": "system", "content": system_prompt},
                        {"role": "user", "content": user_message},
                    ],
                    tokenize=False,
                    add_generation_prompt=True,
                )
            )
        return formatted_prompts
```

C.3 Prompt for LLM Knowledge Extractor

Here is the prompt template for the LLM Knowledge Extractor.

```python
class CragSystem:
    def get_direct_answers(self, queries) -> list[str]:
        system_prompt = """You are provided with a question.
Your task is to answer the question with your reasoning process.
If you can't answer it directly based on your knowledge, respond with 'I don't know'.
If you think the premise of the question is wrong, for example, the question asks information about a person's husband, but you are sure that the person doesn't have one, you should answer with "Invalid question" without any other words.
You **MUST** think if the question has a false premise, then think the final answer.
You **MUST** generate the reasoning process before the answer. You **MUST** generate your output with the following format:

===START===
## Reasoning:
- Does it have a false premise?
<YOUR REASONING>
- What is the final answer?
<YOUR REASONING>
------
## Answer:
<YOUR FINAL ANSWER>
===END===

**IMPORTANT RULES**:
- If you can't answer it directly based on your knowledge, respond with 'I don't know'.
- Your generation **MUST** starts with "===START===" and ends with "===END===".
- `<YOUR FINAL ANSWER>` should be succinct, and use as few words as possible.
- `<YOUR REASONING>` should be a detailed reasoning process that explains how you arrived at your answer.
- If you think the premise of the question is wrong, for example, the question asks information about a person's husband, but you are sure that the person doesn't have one, you should answer with "Invalid question" without any other words.
Let's think step by step now!"""
        formatted_prompts = []
        for _idx, query in enumerate(queries):
            user_message = query
            formatted_prompts.append(
                self.tokenizer.apply_chat_template(
                    [
                        {"role": "system", "content": system_prompt},
                        {"role": "user", "content": user_message},
                    ],
                    tokenize=False,
                    add_generation_prompt=True,
                )
            )
```

C.4 Prompt for KG Module

We show the prompt template of the function-calling method in the knowledge graph module.

```python
class KGToolRAGModel
    def get_tool_references(
        self, queries: list[str], query_times: list[str]
    ) -> list[list[str]]:
        system_prompt = f"""
You are a helpful assistant in function calling. I have a knowledge graph and a set of functions that can be called. You will be given a question and the query time. Your task is to generate several function calls that can help me answer the question. Here are functions and their descriptions:
{TOOLS}
Remember your rules:
1. You **MUST** follow the function signature.
2. You **MUST** output the JSON format that can be read by `json.loads`. Return empty list if no useful function calls can be found.
3. For each function call, you should output its function name and corresponding arguments.

Here are examples:

# Example 1:
Query: which company have larger market cap, hri or imppp?
Query time: 03/13/2024, 10:19:56 PT
Output: [
    {{"function_name": "finance_get_market_capitalization", "args": ["hri"]}},
    {{"function_name": "finance_get_market_capitalization", "args": ["imppp"]}}
]

# Example 2:
Query: who are the current members of the band eagles?
Query time: 03/05/2024, 23:17:59 PT
Output: [
    {{"function_name": "music_get_members", "args": ["eagles"]}}
]
"""
        formatted_prompts = []
        for idx, query in enumerate(queries):
            user_message = f"Question: {query}\n"
            user_message += f"Query time: {query_times[idx]}\n"
            user_message += "Using the tools listed above and based on the question, generate useful function calls for me. \n"
            user_message += """
Remember your rules:
1. You **MUST** follow the function signature.
2. You **MUST** output the JSON format that can be read by `json.loads`.
3. For each function call, you should output its function name and corresponding arguments.
"""
            formatted_prompts.append(
                self.llm.get_tokenizer().apply_chat_template(
                    [
                        {"role": "system", "content": system_prompt},
                        {"role": "user", "content": user_message},
                    ],
                    tokenize=False,
                    add_generation_prompt=True,
                )
            )
```

C.5 Prompt for Reasoning Module

We show the prompt template of the reasoning module, where we successfully controlled the reasoning paths and output format and many corner cases.

```python
system_prompt = """You are provided with a question and various references.
Your task is to answer the question with your reasoning process.
There are also some calculation results from another agent, which may be useful for you.
There is an answer from another agent which may be useful. It may have hallucination. You need to judge whether to trust it by yourself.
If the references do not contain the necessary information to answer the question and you can't answer it directly based on your knowledge, respond with 'I don't know'.
If you think the premise of the question is wrong, for example, the question asks information about a person's husband, but you are sure that the person doesn't have one, you should answer with "Invalid question" without any other words.
You **MUST** think if the question has a false premise, then think the final answer.
You **MUST** generate the reasoning process before the answer. You **MUST** generate your output with the following format:

===START===
## Reasoning:
- Does it have a false premise?
<YOUR REASONING>
- What is the final answer?
<YOUR REASONING>
- Can you answer it based on current knowledge?
<YOUR REASONING>
------
## Answer:
<YOUR FINAL ANSWER>
## False Premise:
<HAS_FALSE_PREMISE_OR_NOT>
===END===

**IMPORTANT RULES**:
- If the references do not contain the necessary information to answer the question and you can't answer it directly based on your knowledge, respond with 'I don't know'.
- Your generation **MUST** starts with "===START===" and ends with "===END===".
- `<YOUR FINAL ANSWER>` should be succinct, and use as few words as possible.
- `<YOUR REASONING>` should be a detailed reasoning process that explains how you arrived at your answer.
- `<HAS_FALSE_PREMISE_OR_NOT>` should be "yes" if the premise is wrong and the question is invalid, and "no" otherwise. It can **ONLY** be chosen from these two options.
- If you think the premise of the question is wrong, for example, the question asks information about a person's husband, but you are sure that the person doesn't have one, you should answer with "Invalid question" without any other words.
Let's think step by step now!"""
user_message = ""
references = ""
if len(retrieval_results) > 0:
    references += "# References \n"
    # Format the top sentences as references in the model's prompt template.
    for _snippet_idx, snippet in enumerate(retrieval_results):
        references += f"- {snippet.strip()}\n"
user_message += f"{references}\n------\n\n"
if len(kg_results) > 0:
    kg_references = ""
    user_message += "## Knowledge Graph references \n"
    for idx, kg_result in enumerate(kg_results):
        kg_references += f"### KG Ref {idx + 1}: \n"
        kg_references += f"{kg_result}\n"
    kg_references = kg_references[:1000]
    user_message += f"{kg_references}\n------\n\n"
    logger.debug("Currect KG references:\n{}".format(kg_references))
if len(related_tables) > 0:
    table_references = ""
    user_message += "## Table references \n"
    for idx, table in enumerate(related_tables):
        table_references += f"### Table {idx + 1}: \n"
        table_references += f"{table}\n"
    table_references = table_references[: self.max_table_length]
    user_message += f"{table_references}\n------\n\n"
if len(generated_expressions) > 0:
    expression_references = ""
    user_message += "## Possible useful calculation results \n"
    for idx, expression in enumerate(generated_expressions):
        expression_references += f"### Calculation {idx + 1}: \n"
        expression_references += f"{expression}\n"
    user_message += f"{expression_references}\n------\n\n"
if direct_answer is not None:
    user_message += (
        f"# An answer from another agent:\n{direct_answer}\n------\n\n"
    )
user_message += """**Remember your IMPORTANT RULES**:
- If the references do not contain the necessary information to answer the question and you
        can't answer it directly based on your knowledge, respond with 'I don't know'.
- Your generation **MUST** starts with "===START===" and ends with "===END===".
- `<YOUR FINAL ANSWER>` should be succinct, and use as few words as possible.
- `<YOUR REASONING>` should be a detailed reasoning process that explains how you arrived at
        your answer.
- `<HAS_FALSE_PREMISE_OR_NOT>` should be "yes" if the question is invalid, and "no" otherwise.
         It can **ONLY** be chosen from these two options.
- If you think the premise of the question is wrong, for example, the question asks
        information about a person's husband, but you are sure that the person doesn't have
        one, you should answer with "Invalid question" without any other words.
"""
user_message += (
    f"Using the references listed above, answer the following question: \n"
)
user_message += f"Current Time: {query_time}\n"
user_message += f"Question: {query}\n"
user_message += "Let's think step by step now!\n"
```

As illustrated in the prompt template, we have devised a set of rules to handle corner cases and enhance the accuracy of the model's responses. We repeat the important rules in both the system prompt and the user prompt to strengthen the control.

C.6 Prompt for Backup Summarization Agent

We show the prompt template of the backup summarization agent below.

```
class SummarizationAgent
    def format_prompts(
        self,
        queries,
        reasoning_answers: list[str] = [],
    ):
        assert len(queries) == len(reasoning_answers)
        system_prompt = "You are provided with a question and a reasoning process from another
                agent. Your task is to summarize the reasoning process and finally answer the
                question succinctly, using the fewest words possible."
        formatted_prompts = []
        for _idx, query in enumerate(queries):
            reasoning_process = reasoning_answers[_idx]
            user_message = ""
            user_message += f"Question: {query}\n"
            user_message += (
                f"# Useful Reasoning Process: \n{reasoning_process}\n-----\n\n"
            )
            user_message += f"Using the reasoning process above, answer the question."
            formatted_prompts.append(
                self.tokenizer.apply_chat_template(
                    [
                        {"role": "system", "content": system_prompt},
                        {"role": "user", "content": user_message},
                    ],
                    tokenize=False,
                    add_generation_prompt=True,
                )
            )
        return formatted_prompts
```