Winning Solution For Meta KDD Cup' 24

Yikuan Xia[*]
wfl00014@pku.edu.cn
Peking University
Beijing, China

Jiazun Chen[*]
chenjiazun@stu.pku.edu.cn
Peking University
Beijing, China

Jun Gao[†]
gaojun@pku.edu.cn
Peking University
Beijing, China

Abstract

This paper describes the winning solutions of all tasks in Meta KDD Cup '24 from **db3** team. The challenge is to build a RAG system from web sources and knowledge graphs. We are given multiple sources for each query to help us answer the question. The CRAG challenge involves three tasks: (1) condensing information from web pages into accurate answers, (2) integrating structured data from mock knowledge graphs, and (3) selecting and integrating critical data from extensive web pages and APIs to reflect real-world retrieval challenges. Our solution for Task #1 is a framework of web or open-data retrieval and answering. The large language model (LLM) is tuned for better RAG performance and less hallucination. Task #2 and Task #3 solutions are based on a regularized API set for domain questions and the API generation method using tuned LLM. Our knowledge graph API interface extracts directly relevant information to help LLMs answer correctly. Our solution achieves 1st place on all three tasks, achieving a score of 28.4%, 42.7%, and 47.8%, respectively.

CCS Concepts

• **Computing methodologies → Natural language generation**.

Keywords

Large Language Models, RAG

1 Introduction

Ensuring the trustworthiness of language model (LLM) responses is critical due to the persistent issue of hallucination, where models generate inaccurate or ungrounded answers. Studies show that GPT-4's accuracy for fast-changing facts is often below 35% [8]. Retrieval-Augmented Generation (RAG) [1, 4] offers a promising solution by integrating external information retrieval with LLMs to provide grounded answers. Despite its potential, RAG faces challenges in selecting relevant information, reducing latency, and synthesizing complex answers.

To address these issues, Meta introduces the Meta Comprehensive RAG Challenge (CRAG) as a 2024 KDDCup event, aiming to benchmark RAG systems with clear metrics and evaluation protocols to drive innovation and advance solutions in this field. CRAG Benchmark encompasses five domains, eight question types, varying answer timelines, and a range of entity popularity, including head, torso, and tail facts, as well as simple and complex question formats to test reasoning and synthesis capabilities. Each query has a time budget of 30 seconds, which also poses an efficiency challenge to the candidate solutions.

In detail, the CRAG challenge consists of three tasks:

(1) **Task #1: Web-based Retrieval Summarization.** There are five web pages per question to identify and condense relevant information into accurate answers.
(2) **Task #2: Knowledge Graph and Web Augmentation.** Mock APIs are provided to access structured data from mock knowledge graphs to integrate information into comprehensive answers.
(3) **Task #3: End-to-End RAG.** 50 web pages and Mock APIs access per question are provided to select and integrate the most important data, reflecting real-world information retrieval challenges.

The author's team, db3, participates in the contest and achieves first place in the three tasks, gaining a score of 28.4%, 42.7%, and 47.8%, respectively. This paper describes the author's solution to the three tasks. The difference between the three challenges is that the information sources are different. Since Task #2 and Task #3 are provided with both sources, we describe the solution of these two tasks together. Our code is available on GitLab [1].

In the remainder of the paper, we discuss the web retrieval module and related model adjustment in Sec. 2, and we discuss the knowledge graph extraction module and related model adjustment in Sec. 3. We conclude our work and look into future works in Sec. 4.

2 Solution to Task #1

In this section, we will propose our solution to Task #1. Specifically, we will present the pipeline of processing web pages, which is also used in Task #3. We will also describe the pipeline of tuning the base LLM, which may be similar for other purposes in Task #2 and Task #3.

2.1 Framework for Solving Task #1

Figure 1 illustrates our framework for Task 1. We employ two pathways to retrieve information and ultimately use a tuned LLM to answer the questions, which relieves the hallucination problem. Next, we will separately introduce each pathway and the adjustments made to the LLM.

2.2 Web Retrieval Pathway

We follow the widely used retriever and reranker framework for our web retrieval pathway.

Text Extraction from Web Pages. For the retrieval process, the text extraction module processes the HTML contents of search results, converting them into plain text chunks. Specifically, we use the BeautifulSoup library[2] to extract text content from raw HTML.

[*]Both authors contributed equally to this research.
[†]Corresponding Author

[1]https://gitlab.aicrowd.com/jiazunchen/kdd2024cup-crag-db3
[2]https://www.crummy.com/software/BeautifulSoup/

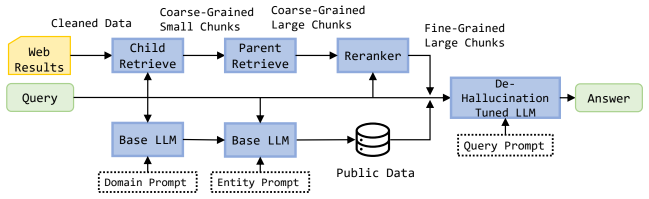


Figure 1: Illustration of Task 1 framework.

We use the CharacterTextSplitter from LangChain[3] to split the text into chunks.

Parent-Child Chunk Retriever. The Retriever ranks the text chunks by calculating the similarity between the query and each text chunk. Specifically, the retriever analyzes the keywords along with semantics within the query and matches them with the content of the text chunks. We refer to the open-source retriever leaderboards for retriever selection to ensure the most effective retriever is used. The final candidates are bge-base-en-v1.5 [7] and bce-embedding-base_v1 [6], producing similar results. The submitted version is based on bge-base-en-v1.5.

Since relatively smaller chunks have better retrieval precision and relatively larger chunks retain more information, we use the ParentDocumentRetriever from LangChain to manage a parent-child chunk split. We use the relatively small child chunks to retrieve from the question and the ParentDocumentRetriever to maintain the inclusion relationship. The smaller child chunks, such as individual sentences, are used for retrieval, while the larger parent chunks, which contain these retrieved child chunks, are typically whole paragraphs and are fed into the RAG system.

In the contest, larger parent chunk size will result in a larger context for LLM, leading to more inference time. So, to balance the time issue, we select parent_chunk_size=700, child_chunk_size=200 as a baseline. Under the submission constraints, we tune these hyperparameters, finding that parent_chunk_size=500,1000,2000 is also acceptable.

Reranker. The results from the retriever can be regarded as a coarse selection. In this challenge, we set the retriever to return *recall_k* text chunks. These chunks contain potentially relevant information, but not all are equally important or relevant.

To more effectively utilize this information within the limited context, we introduce a reranker model. The reranker model performs a secondary screening by more finely evaluating and ranking the *recall_k* data chunks returned by the Retriever. Through this detailed screening process, we can identify the most valuable and relevant chunks, ultimately selecting the top *reranker_k*. These top *reranker_k* chunks will serve as the basis for further processing and answering the questions, ensuring that the our answers are more accurate and reliable.

We refer to the open-source reranker leaderboards for reranker selection. The final candidates are bge-reranker-v2-m3 [2, 5] and bce-reranker-base_v1 [6]. The results are rather similar. The submitted version is based on bge-reranker-v2-m3.

[3]https://www.langchain.com/

To fit in the LLM context and given a limited running time, we set the recall number for the retriever *recall_k* = 50. For the number of chunks we fed to the LLM *reranker_k*, we set it according to the parent_chunk_size. For example, we take *reranker_k* = 5 if parent_chunk_size=2000, and we would take *reranker_k* = 10 if parent_chunk_size=1000.

2.3 Public Data Pathway

A regular web retriever usually suffers greatly from insufficient information and misinformation. For some of the stable facts, we can gather public data to provide additional information. This information is preprocessed into a paragraph, which is combined with the content of the web search to help the LLM answer the question.

This information for different domains is constructed differently. For the movie domain, we preprocess the Oscar award information[4] and the Full MovieLens[5]. For finance, we preprocess the current pe-ratio, market cap and eps stats for every current stock in America. For music, we preprocess the Grammy award information[6]. Due to the limited information in other domains, no preprocessing is conducted. Specifically, we serialize the entity's table/json data into natural language format using the structure "The [entity attribute] is [value of the entity attribute]." Specifically, we serialize the entity's table/json data into natural language format using the structure "The [entity attribute] is [value of the entity attribute]." For example, for a specific movie, the preprocessed format is: *The title is "Rain Man." The director is Barry Levinson. The lead actors are Dustin Hoffman and Tom Cruise. The release year is 1988...*

We store the preprocessed information using the corresponding entities as keys and then utilize LLM to locate the query entities through in-context learning. Specifically, we first locate the problem domain based on **Domain Prompt**:

> [{"role": "system", "content": "You are an assistant expert in movie, sports, finance and music fields."},
> {"role": "user", "content": "Please judge which category the query belongs to, without answering the query. You can only and must output one word in (movie, sports, finance, music). If the question doesn't belong to movie, sports, finance, music, please answer other.
> Question: {query_str}
> Answer: "}].

Because open domain questions often include content from other domains, we use "other" instead of "open". Next, we design different prompts for various domains to query the entities. For example, for movie **Entity Prompt**:

> [{"role": "system", "content": "You are a helpful and honest assistant. Please, respond concisely and truthfully in {token_limit} words or less. If you are not sure about the query,

[4]https://www.kaggle.com/datasets/unanimad/the-oscar-award
[5]https://www.kaggle.com/datasets/rounakbanik/the-movies-dataset?select=movies_metadata.csv
[6]https://www.kaggle.com/datasets/unanimad/grammy-awards

```
answer I don't know. There is no need to explain the reasoning behind your answers. "},
{"role": "user", "content": Given a query about movies, return the title of each movie in below formats.
If multiple movie names are involved, connect with '&&'.
#Examples:
Question: which movie was created first, a walk to remember or the notebook?
Answer: a walk to remember && the notebook
......
#Query:
Question: {query_str}
Answer: "}].
```

Regarding how the entities in LLM's responses are linked to entities in the dataset, the required level of matching varies based on the characteristics of the question domain. Matching criteria range from strict to lenient, including exact character matches, substring inclusion, or similarity comparisons using embedding models.

2.4 LLM Inference Module

According to the contest rules, the correct answer will be awarded 1 point, and the incorrect answer will be penalized for 1 point. So, a significant challenge during the contest is reducing hallucinations and dealing with invalid questions. We will present our LLM inference module to the two challenges.

Base model. We follow the contest instructions to use the LLama series LLM [7]. Considering the limited running time, we use the Llama-3-8B-instruct model as the base model.

Basic Query Prompt. We use the following basic prompt p_{basic} to generate answers:

```
[{"role": "system", "content": "You are a helpful and honest assistant. Please, respond concisely and truthfully in {token_limit} words or less. Now is {query_time}"},
{"role": "user","content": "Context information is below.
{context_str}
Given the context information and using your prior knowledge, please provide your answer in concise style. End your answer with a period. Answer the question in one line only.
Question: {query_str}
Answer: "}]
```

where token_limit is the limit to the answer that we want to control (as an answer longer than 75 tokens will be truncated.), query_time is the time when the question is asked, which is crucial in real-time questions, context_str is formed by combining data from public retrieval and $reranker_k$ web retrievals using <doc> tokens, then truncated based on a maximum token limit of 4000 (note that the previous retriever's chunk size is based on characters, while here it is based on tokens), query_str is the query.

Reduce Hallucination using Prompt Control. Hallucination can partly be controlled by prompt design. Certain instructions in the prompt can hold the LLM from generating wrong facts. E.g., we can use the following controlled prompt p_{ctrl} to generate higher-quality answers:

```
[{"role": "system", "content": "You are a helpful and honest assistant. Please, respond concisely and truthfully in {token_limit} words or less. Now is {query_time}"},
{"role": "user","content": "Context information is below.
{context_str}
Given the context information and using your prior knowledge, please provide your answer in concise style. Answer the question in one line only.
If the question is based on false prepositions or assumptions, output "invalid question". For example, What's the name of Taylor Swift's rap album before she transitioned to pop? (Taylor Swift didn't release any rap album.)
If you are not sure about the question, output "i don't know"
Question: {query_str}
Answer: "}]
```

Using p_{ctrl} to control the generation can lead to better results, as for questions hard for the LLM to answer, it will probably avoid the penalty. For some invalid questions, the tuned model may point out that the questions are based on false prepositions. However, the overall results are not satisfying. So, we don't directly use this method to control hallucination.

Reduce Hallucination from Fine-tuning. Usually, through sufficient supervised fine-tuning (SFT), it's possible to make the LLM perform better on a particular task. So, we try fine-tuning the base model to reduce hallucination further in the contest.

Through experiment, we find out that using p_{ctrl} to generate answers will reduce hallucination, while in the meantime, some questions which can be answered correctly using p_{basic} will be answered wrongly with "i don't know". So, we hope we can leverage the whole potential of the RAG system while hindering most of the wrong answers.

It's clear that for some continuing changing facts, it's impossible for the LLM to answer correctly if not provided with the fact in the context_str. Our intuition is that we hope the LLM can answer correctly for the facts contained in the context_str. For the facts that are not contained in the context_str but in the LLM's internal knowledge, we hope the LLM can answer correctly, too. For the other queries, which are out of the RAG's capabilities, we hope the LLM can answer "i don't know" honestly. As common sense, such patterns may be learned by the LLM. For example, for the continuing changing finance problems, the answer accuracy is close to 0, so the LLM may learn to answer honestly "i don't know" for such questions.

So, we follow the following steps to generate the labels for SFT to meet our intuition:

- 1. For the queries identified in the ground truth as invalid questions, we set the labels to "invalid question", hoping that the model can possess the capability to find questions based on false premises.
- 2. We generate answers using the p_basic prompt for each query in the training split to leverage the potential of our RAG system fully. We use prompt p_{check_gt} and LLM to

[7]https://llama.meta.com/

judge whether the answer is correct. (The LLM here can be Llama3, or, more accurately, GPT4.[8])

- 3.1 For the queries labeled as correctly answered, we label the query with the ground truth answer because the query is within the potential of the current RAG system.
- 3.2 For the queries labeled as wrongly answered, we use LLM to check whether the ground truth answer can be inferred from the context_str using $p\_context$. (The LLM here can be Llama3 or, more accurately, GPT4.)
- 3.2.1 For the queries with context_str that the LLM regards as irrelevant to the ground truth answer, we label these queries with "i don't know", as these questions can hardly be answered correctly.
- 3.2.2 For the queries with context_str that the LLM regards as relevant to the ground truth answer, we label these queries with the ground truth answer. We hope the fine-tuned LLM can have a stronger comprehension ability from the noisy context containing hints of the correct answer.

We move the fine-tuning part after constructing the training data as above. Considering limited computation resources, we use LoRA [3] to fine-tune the base model. Using LoRA models has another advantage. We can fine-tune several LoRA models each responsible for a subtask. Since LoRA parameters are tiny compared with the LLM parameters, it's easy to switch between the subtasks, which is time-saving. We tune the basic model for 2-3 epochs on the training set. The tuning hyperparameters can be found in the appendix. After tuning, there are three significant improvements:

1. The answer style becomes closer to the short and direct answer as the ground truth answers, making it harder for the LLM to generate hallucinations during reasoning.
2. The LLM can judge the "i don't know" case more accurately, leading to a better result.
3. The LLM can deal with some false premise cases, saving many points.

Inference Acceleration. We use vLLM to accelerate the inference process [9]. However, though the latest vLLM library supports the LoRA framework, the graphics card driver in the test environment has some compatibility issues. So, we have to load each LoRA model completely when we aim to switch between multiple LoRA models. Each query can share the switching time through batch inference to meet the time requirement. Our submission contains versions that use and do not use vLLM. If the compatibility issue is solved, introducing vLLM can save much time.

3 Solution to Task #2 and Task #3

In this section, we will first introduce the main framework of our solution for Task #2 and Task #3. Then, we will propose our knowledge graph retrieval module based on a set of regularized APIs and API generation using a tuned LLM. The web retrieval and answer generation part of Task #2 and Task #3 is similar to the ones in solution to Task #1, which will be presented briefly in this section.

[8]For long answers generated by Llama3, it appears that judging also by Llama3 would be inaccurate. The answer may be more favored by its generated model. As for short answers, this problem is relieved greatly.
[9]https://github.com/vllm-project/vllm

3.1 Framework for Solving Task #2 and Task #3

We observe that the information extracted from web pages is noisier than the information from the Mock APIs. Therefore, our framework separates the answer from the Mock API and web pages, and we prioritize the results based on the Mock API results. Once the results based on the Mock API content are not "i don't know", we accept the result and output directly. The framework of Task #2 and #3 is illustrated in Fig. 2.

Notably, the web retrieval part for Task #3 is slightly different from those in Task #1 and Task #2. Since Task #3 has 50 web pages, the time budget should be taken into consideration. In the contest, a page snippet is provided for each web page. We use the reranker to find the top five related web pages based on the snippets, which makes the overall running time for the retriever acceptable, and then process them according to the method used in Task 1. Additionally, we don't use public data in Task #2 and #3 because it's basically covered by Mock APIs.

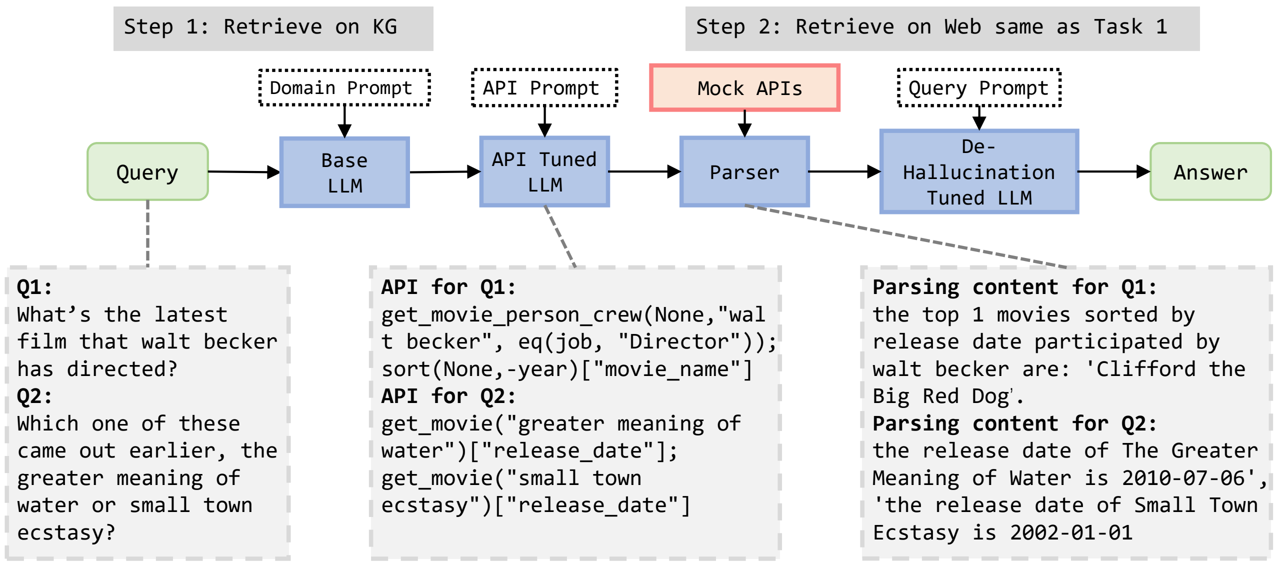
3.2 Knowledge Graph Retrieval Module

A Mock API knowledge graph is provided in Task #2 and Task #3. We will describe how we extract useful information from this API or knowledge graph. The main idea of our knowledge graph retrieval module is to use LLMs to generate a series of API calls, which extract the specific information the query is asking about.
Regularization of the API. Though the APIs provided contain rich information, it's quite hard to locate the exact information we want. The exact location of the information can be found by executing the code generated by the LLM. However, the code generation capability of a local 7B LLM is rather limited. So we design a regularized version of API. For each query, only one (or several) APIs are generated. From the generated regularized APIs, we use a parsing system to get the generated results. The results are then converted to natural language to form the output of our retrieval module.

We take the movie domain as an example. The original movie API consists of the following API calls:

- get_person_info (person_name)⟶ name (string): name of person; acted_movies (list); directed_movies (list); birthday (string); oscar_awards
- get_movie_info (person_name)⟶ title (string); release_date (string); original_title (string); original_language (string); budget (int); revenue (int); rating (float); genres (list); oscar_awards; cast (list); crew(list)
- get_year_info (year)⟶ movie_list(list); oscar_awards(list)

This actually forms a relational database behind the scenes. There are two relational tables: the PERSON table and the MOVIE table. For instance, the PERSON table has the columns: name, birthday, and the MOVIE table has the columns: title, release_date, original_title, original_language, budget, revenue, rating, genres, year. There are three additional tables that have foreign keys referring to the PERSON table and MOVIE table, which records the relationship between person and movies: the CAST table, the CREW table and the OSCAR table. Each entity of the five tables can be constructed using the API calls. So theoretically, we can use SQL language to query information from this relational database, and converting the query to SQL language is a typical text2sql task [9], which is



```
Q1:
What's the latest film that walt becker has directed?
Q2:
Which one of these came out earlier, the greater meaning of water or small town ecstasy?
```

```
API for Q1:
get_movie_person_crew(None,"walt becker", eq(job, "Director"));
sort(None,-year)["movie_name"]
API for Q2:
get_movie("greater meaning of water")["release_date"];
get_movie("small town ecstasy")["release_date"]
```

```
Parsing content for Q1:
the top 1 movies sorted by release date participated by walt becker are: 'Clifford the Big Red Dog'.
Parsing content for Q2:
the release date of The Greater Meaning of Water is 2010-07-06',
'the release date of Small Town Ecstasy is 2002-01-01
```

Figure 2: Illustration of Task #2 and #3 framework.

well studied. However, it's impossible to call the API multiple times to form a relational database and execute SQL on it due to time budget. So, we design a new type of regularized API that is easy to parse and execute and leverages the characteristics of relational databases.

For the movie domain, we set the following API:

- get_person (person_name,condition)[key_name], which is equal to *SELECT key_name FROM PERSON WHERE condition and name=person_name*
- get_movie(movie_name,condition)[key_name], which is equal to *SELECT key_name FROM MOVIE WHERE condition and title=movie_name*
- get_movie_person_X(movie_name,person_name,condition) [key_name], (here X=CREW, CAST or OSCAR), which is equal to *SELECT key_name FROM MOVIE, PERSON, X WHERE condition and title=movie_name, name=person_name.*

Using these three regularized APIs, most of the information the query wants in this contest can be extracted with only a few API calls. These APIs can also be easily evaluated by the original APIs through a parsing system. More importantly, these APIs are easy to generate for the LLM, as essentially, only template selection and extraction of entity names have to be done by the LLM. There are no complex codes or SQL generation involved, which may lead to better performance.

We design different APIs for the five different domains, but the design choices are all similar. The details of these APIs can be found in our released codes.

Details of the API System. How to make the conditions work in the above system is a question. Answering some of the questions in the contest involves a slight modification of the API system above. For instance, querying the latest Spielberg film actually requires the sort function. For some queries in the post-processing questions, numeric computations are involved. To meet these requirements, we have some other words for generation in our designed corpus:

- We can use cmp (key_name,value_name) to set a condition. The cmp here can be neq, eq, ge and le, which represents not equal, equal, greater, less respectively. e.g., eq(gender,male), which means the condition of gender to be male. The condition can be a list of multiple conditions , e.g., [eq(gender,"male"),eq(character,"batman")].
- We can use ["len"] to get the output lengths of a result, and we can use AVG to get the average numeric value of the output.
- We can use sort(condition,sort_key_name) to get a sorted list which satisfies such condition, and the list is sorted using the sort_key_name. If we want descending sort, we can use -sort_key_name.

These are all simplified operators for coded functions or functions in SQL. Other functions include using * to represent the output of the last query and using * in another query, which is similar to the sublist function in SQL. This can perfectly fit the multi-hop query scenario. It's a pity that we haven't finished this part due to limited time in this contest, and we look forward to developing such functions in future systems.

Parser. We manually program the parser to meet the requirements of most queries in the development set. This may affect the generalization capability of this system, so using the standard SQL execution engine may be a better choice if the relational databases are presented. The output of the parser is converted to natural language so that even if irrelevant information is extracted, the LLM may be aware of that and refuse to answer.

Examples of New Regularized APIs. Here are three ideal examples of the new regularized API in the movie's domain:

- Which one of these came out earlier, *the greater meaning of water* or *small town ecstasy*? → get_movie("*greater meaning of water*")["release_date"]; get_movie("*small town ecstasy*")["release_date"]
- Who won the best actor oscar for their performance in a movie in 2012? →get_movie_person_oscar (None,None,[eq(year,2012),eq(category,"best actor") ,eq(winner,"true")])["name"]
- What's the latest film that *walt becker* has directed? → get_movie_person_crew(None,"*walt becker*", eq(job, "Director")); sort(None,-year)["movie_name"]

An example of the full pipeline can also be found in Fig. 2.

API Generation. After setting down the rules and parser of the new regularized API system, we move to solving the API generation problem. We use prompts to help the LLM generate as many valid APIs as possible to help us extract information. Neglecting the system message, the API generation prompt p_{gen_API} is as follows:

```
You are given a query about movies, and several APIs to get
information from a database How to collect useful information from the database using the given APIs.
The schema of entities is as follows:
{Schema_info}
The API rules are below:
{API_rules}
Here are some examples:
{ICL_examples}
Generate the answer only using the information from the
query. Please strictly follow the format in the examples and
APIs, you do not have to provide the code, only the use of API
in the examples. The only allowed format is multiple lines
of get_X,sort. (sort is optional) Please complete the answer
only:
Query:{query_str}
Answer:
```

where Schema_info are some descriptions about the underlying relational database schema, restricting the LLM to generate key names in the relational tables, API_rules are the rules for generating the API described in the above subsection, query_str is the question, and ICL_examples are some in-context learning selected examples of query and API pairs.

Here, the in-context learning examples are selected manually iteratively. First a few examples are selected, and we generate 100 examples using the LLM. Then the queries with the wrongly generated examples are added to the examples. Notably, selecting relevant ICL examples may be extremely effective in this scenario, as for similar queries, only the entity names have to be substituted. We haven't realized this function, but we believe it may have great results. Details of this prompt for different domains can be found in our source code. Here, we present some details of the prompt for movies in the Appendix.

Fine-tuning for API Generation. Through experiment, we observe that using p_{API_gen} still requires the LLM to have relatively strong capabilities. Strong LLMs, e.g., GPT-4, perform better than the local Llama3 8B. So we hope fine-tuning the local LLM can help boost the performance of API Generation. For the fine-tuning ground truth data, We use GPT-4 and p_{API_gen} to generate a first version of ground truth APIs for convenience. Then, we manually label the ground truth APIs for higher quality. We also use LoRA to fine-tune our base Llama3 model, as we need to efficiently switch between different LoRA parameters under the time budget. We also fine-tune the base model for 2-3 epochs, and the hyperparameters are also listed in the Appendix.

3.3 LLM Inference Module

We follow the same framework in Sec. 2 to fine-tune the model for LLM inference. The tuned model has a higher priority than the model tuned on web page results.

4 Conclusion

The Meta KDDCup 24 competition is a unique challenge due to the various types of information sources and the changing facts, which are difficult for the LLMs. We have presented how we addressed these challenges successfully in all three tasks of the contest. Our solution for Task #1 is a framework of web or open-data retrieval and tuned LLM for question answering. The solution to Task #2 and Task #3 is based on a regularized API set for domain questions and the API generation method using tuned LLM. We will further look into the balance of efficiency and effectiveness in RAG and a refined API parsing system for RAG to extract information from structured sources in the future.

5 Acknowledgement

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant No. 62272008.

References

[1] 2024. *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net.
[2] Jianlv Chen, Shitao Xiao, Peitian Zhang, Kun Luo, Defu Lian, and Zheng Liu. 2024. BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation. arXiv:2402.03216 [cs.CL]
[3] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685* (2021).
[4] Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). 2020. *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual.*
[5] Chaofan Li, Zheng Liu, Shitao Xiao, and Yingxia Shao. 2023. Making Large Language Models A Better Foundation For Dense Retrieval. arXiv:2312.15503 [cs.CL]
[6] Inc. NetEase Youdao. 2023. BCEmbedding: Bilingual and Crosslingual Embedding for RAG. https://github.com/netease-youdao/BCEmbedding.
[7] Shitao Xiao, Zheng Liu, Peitian Zhang, and Niklas Muennighoff. 2023. C-Pack: Packaged Resources To Advance General Chinese Embedding. arXiv:2309.07597 [cs.CL]
[8] Xiao Yang, Kai Sun, Hao Xin, Yushi Sun, Nikita Bhalla, Xiangsen Chen, Sajal Choudhary, Rongze Daniel Gui, Ziran Will Jiang, Ziyu Jiang, Lingkun Kong, Brian Moran, Jiaqi Wang, Yifan Ethan Xu, An Yan, Chenyu Yang, Eting Yuan, Hanwen Zha, Nan Tang, Lei Chen, Nicolas Scheffer, Yue Liu, Nirav Shah, Rakesh Wanga, Anuj Kumar, Wen tau Yih, and Xin Luna Dong. 2024. CRAG – Comprehensive RAG Benchmark. arXiv:2406.04744 [cs.CL] https://arxiv.org/abs/2406.04744
[9] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. *arXiv preprint arXiv:1809.08887* (2018).

A Detail Evaluation of Our Solution

The organizer presented the detail evaluation of our solution, and we list it in Fig. 3. As we can see extracting useful information from KGs can significantly boost our performance on real-time and fast-changing queries. Accuracy boost on the finance domain is also sound, even outperforming some commercial RAG systems [8].

B LoRA and FineTuning Hyperparameters.

The LoRA and Finetuning hyperparameters used in tuning the base model and API generation module is listed in Tab. 1:

Table 1: LoRA and FineTuning Hyperparameters.

Name	Value
LoRA_alpha	16
LoRA_dropout	0.1
LoRA_r	8
target_modules	["k_proj", "q_proj", "v_proj", "up_proj", "down_proj", "gate_proj"]
bias	"none"
4-bit	True
max_seq_length	2048/4096
per_device_train_batch_size	1
gradient_accumulation_steps	4
optim	"adamw_hf"
learning_rate	2e-4
max_grad_norm	0.3
scheduler	"cosine", warm_up_ratio=0.1

C Used Prompts.

C.1 Prompt for checking the correctness of an answer.

Neglecting the system message, p_{check_gt} is as follows:

INSTRUCTIONS =
# Task:
You are given a Question, a model Prediction, and a list of Ground Truth answers, judge whether the model Prediction matches any answer from the list of Ground Truth answers. Follow the instructions step by step to make a judgement.
1. If the model prediction matches any provided answers from the Ground Truth Answer list, "Accuracy" should be "True"; otherwise, "Accuracy" should be "False".
2. If the model prediction says that it couldn't answer the question or it doesn't have enough information, "Accuracy" should always be "False".
3. If the Ground Truth is "invalid question", "Accuracy" is "True" only if the model prediction is exactly "invalid question".
# Output:
Respond with only a single JSON string with an "Accuracy" field which is "True" or "False".
# Examples:
{ICL_examples}
# Query:
Question: {query_str}
Ground truth: {gt_str}
Prediction: {our_str}
Accuracy:

where ICL_examples are some in-context learning examples for the query, gt_str is the ground truth answer, and our_str is our generated answer.

C.2 Prompt for Checking the RAG Context.

Neglecting the system message, $p_{context}$ is as follows:

We have the following context information:
{context_str}
We have a question: {query_str}
The ground truth answer is: {gt_str}
Is the ground truth answer mentioned in the context information? Answer with yes or no.

where query_str is the question, gt_str is the ground truth answer, and context_str is the context from our retrieval system.

C.3 Prompt for API Gereration (Movie)

Neglecting the system message, p_{API_gen} for the movie domain is as follows:

You are given a query about movies, and several APIs to get information from a database How to collect useful information from the database using the given APIs.
The schema of entities are as follows:
Movie:- title (string): title of movie
...
- year (string): year of the movie
Person:- name (string): name of person
- birthday (string): string of person's birthday, in the format of "YYYY-MM-DD"
Besides we have the concat tables for the concat of these two basic entities:
Cast Movie Person: list of cast members of the movie and their roles. The schema of the cast member entity is:
-'movie_name':name of the movie,
...
-'year'(string):the year of casting
Crew Movie Person: list of crew members of the movie and their roles.
-'movie_name':name of the movie,
...
-'year'(string):the year of crewing
Oscar info: list of oscar awards, win or nominated, in which the movie was the entity. The schema for oscar award entity are:
'year' (int): year of the oscar ceremony,
...

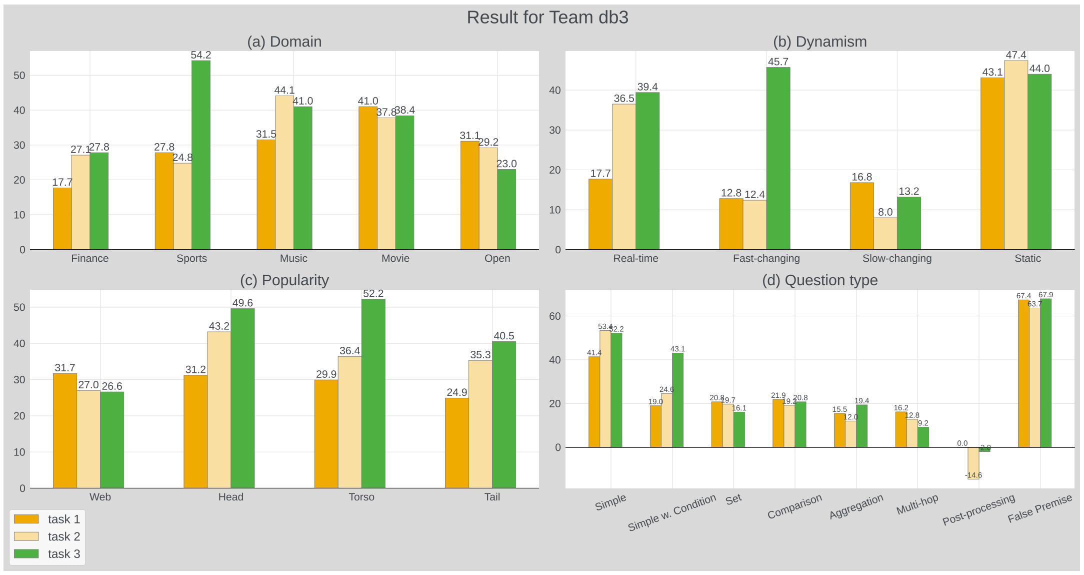


Figure 3: Detail Evaluation of Team db3 Solution

'winner' (bool): whether the person won the award

The API rules are below:
1.you can use cmp(key_name,value_name) to set a condition, the cmp here can be neq,eq,ge,le, which represents not equal,equal, greater, lesser respectively. e.g eq(gender,male), which means the contion of gender to be male, ge(revenue,10), which means the condition of revenue greater than 10. the condition can be a list of multiple conditions,
e.g. [eq(gender,"male"),eq(character,"batman")] you can add condition to the last parameter of get_X_info(X_key_value, condition)
2.you can use get_movie(movie_name,condition)[key_name] to search movie_name for the most relevant result under such condition and query the key_name attribute of it. the key names valid to use with get_movie_info is the key of the movie entities.
...
8.By default we output the first element of one list, however if you want it all, you can add ALL in the front of the command, e.g. ALL, ALL get_movie_person_crew("batman","Jack",1997), represent get ALL Jack crews of batman movies in 1997. You can use [:n] to represent take the first n result of the list

Here are some examples:
{ICL_examples}
Generate the answer only using the information from the query. Please strictly follow the format in the examples and APIs, you do not have to provide the code, only the use of API in the examples. The only allowed format is multiple lines of get_X,sort. (sort is optional) Please complete the answer only:
Query:{query_str}

Answer: