Enhancing LLM Tool Use with High-quality Instruction Data from Knowledge Graph

Anonymous ACL submission

Abstract

Teaching large language models (LLMs) to use tools is crucial for improving their problem-solving abilities and expanding their applications. However, effectively using tools is challenging because it requires a deep understanding of tool functionalities and user intentions. Previous methods relied mainly on LLMs to generate instruction data, but the quality of these data was often insufficient. In this paper, we propose a new method that uses knowledge graphs to generate high-quality instruction data for LLMs. Knowledge graphs are manually curated datasets rich in semantic information. We begin by extracting various query pathways from a given knowledge graph, which are transformed into a broad spectrum of user queries. We then translate the relationships between entities into actionable tools and parse the pathways of each query into detailed solution steps, thereby creating high-quality instruction data. Our experiments show that LLMs fine-tuned with this data significantly improve their tool utilization and overall capabilities.

1 Introduction

The use of tools is a revolutionary hallmark of advanced intelligence in human civilization (Qin et al., 2024), deeply expanding the limits of our physical capabilities. Integrating real-world tools with powerful large language models (LLMs) is imperative to unleash their problem-solving potential in accuracy, efficiency, and automation.

However, proficiently manipulating tools remains a challenging task for LLMs because it requires a thorough understanding of tool functionalities and a deep insight into varying user intentions. Recently, some studies have found that instruction tuning can significantly enhance the tool-use capabilities of LLMs (Tang et al., 2023; Qin et al., 2023; Liu et al., 2024). These methods typically start by collecting real APIs or synthesizing simulated APIs. They then use LLMs to generate user queries based on the APIs. After that, LLMs are employed again to synthesize detailed solution steps for each query, including tool invocations.

Despite the progress made by these methods, several limitations in the construction of instruction data may hinder their full potential, including unguaranteed data quality, insufficient query complexity, and prohibitive costs. First, data quality is crucial for instruction-tuning-based methods and directly impacts the capabilities of LLMs (Gunasekar et al., 2023; Zhou et al., 2024a). Researchers often use advanced commercial models like ChatGPT or GPT-4 to generate data. However, even with these models or careful human review, errors can still occur in the dataset. Second, many methods (Qin et al., 2023; Srinivasan et al., 2023) randomly sample APIs and prompt LLMs to generate queries. This simple approach often leads to irrelevant tool combinations and low-complexity queries rather than high-complexity ones that require advanced reasoning skills. As a result, the final dataset may not be challenging enough to fully engage the reasoning and planning capabilities of LLMs. Moreover, the inconsistent quality of LLM-generated data necessitates meticulous manual review and rigorous revision. The extensive human intervention throughout the data construction process—from initial creation to final validation—makes rapid scaling impractical and labor costs prohibitive.

In this paper, we tackle these challenges by using knowledge graphs (KGs) to create high-quality instruction-tuning data. KGs contain rich, structured knowledge with concepts and relationships represented as nodes and edges. From a tool use perspective, the basic unit of KGs—the "entity-relation-entity" triple—can be interpreted as "input-function-output." By extracting subgraphs from KGs, we can generate complex tool combinations that exhibit **high complexity**, along with corresponding queries and solution paths. Since KGs are carefully curated and verified by humans, their

accuracy and reliability are well-established, ensuring the integrity of the extracted subgraphs. By converting these subgraphs into natural language through a simple formatting process, we can easily generate tool-using instructions and solution paths without relying on potentially flawed LLM-generated data. This approach avoids errors and noise, maintaining the **high quality** of our datasets. Moreover, our method bypasses labor-intensive prompting and eliminates redundant interactions with LLMs. Leveraging existing large-scale KGs and using diverse sampling patterns without manual verification, our approach provides an efficient, **low-cost** solution for scaling up datasets.

In particular, we present a new framework that generates high-quality instruction data by leveraging query-solution pairs from KGs. Our approach integrates First-Order Logic (FOL) queries into the data generation process, ensuring precise execution of each step and guaranteeing answer quality. The framework extracts subgraphs from KGs that match predefined FOL patterns, representing tool utilization queries and solution paths. By executing API sequences associated with these queries, we log API calls and outcomes, creating solution paths and an instruction-tuning dataset called ToolKG. By fine-tuning various LLMs with ToolKG, we observe significant performance improvements on the T-Eval benchmark. Our framework thus provides a high-quality, low-cost solution for enhancing LLM tool utilization.

Our major contributions are as follows:

- We propose to utilize knowledge graphs to generate high-quality instruction data to enhance LLM tool use capability.
- We design a new framework that utilizes FOL queries as intermediates to transform data in KGs into tool-use format, including the generation of APIs, queries, and solution paths.
- We conduct extensive experiments with various LLMs that validate the effectiveness of our synthesized data.

2 Related Work

2.1 Tool Use of LLMs

Integrating external tools within LLMs has emerged as a growing field of research (Qin et al., 2024). Current methodologies can be delineated into two discrete lines. The first line of methods stimulates the tool-use capabilities within LLMs via pure prompting strategies, enabling full interactions among language models, users, and tools. These endeavors encompass the utilization of LLMs with a diverse array of tools, such as code interpreters (Gao et al., 2023), search engines (Yao et al., 2022), retrieval frameworks (Khattab et al., 2022), etc. (Shen et al., 2024; Lu et al., 2024). The remaining methods in the second line enhance tool utilization abilities within LLMs using supervised fine-tuning (SFT). They commonly leverage closed-sourced LLMs like ChatGPT to construct instruction-tuning datasets tailored for tool usage.

Meanwhile, Retrieval-Augmented Generation (RAG) (Lewis et al., 2020) techniques have also been integrated into studying tool learning within LLMs. While some may opt to employ retriever to enhance prompting method (Yuan et al., 2023) directly, other approaches incorporate retrieval components and tool-use tuning procedures via API retrieval (Qin et al., 2023) or prompt demonstration (Srinivasan et al., 2023).

2.2 Tool-use Instruction Dataset

Instruction tuning relies heavily on curated datasets to enhance LLMs' capacity to comprehend human instructions and generate appropriate responses (Wei et al., 2021; Bach et al., 2022). Thus, constructing instruction data is the core of tool learning methods based on instruction tuning. The construction process usually consists of three phases: API collection, query generation, and solution path annotation. ToolAlpaca (Tang et al., 2023) simulates an environment to generate tool-use instances without manual intervention. ToolFormer (Schick et al., 2024) meticulously designs a bootstrapping framework including in-context learning (ICL) prompting, API calls sampling, executing, and filtering to generate an interleaved dataset with API invocations. Llama3 (Dubey et al., 2024) utilizes a combination of human preference annotations and manual rewrites progressively to generate annotation data. However, these data construction methods inevitably either involve costly human annotations or heavily rely on unreliable LLM generation, leading to high expenses or unguaranteed quality. Our approach, in contrast, could address these issues.

2.3 KG for LLMs

Incorporating KGs can significantly enhance LLMs' ability to acquire up-to-date information and factual knowledge, thereby reducing hallucina-

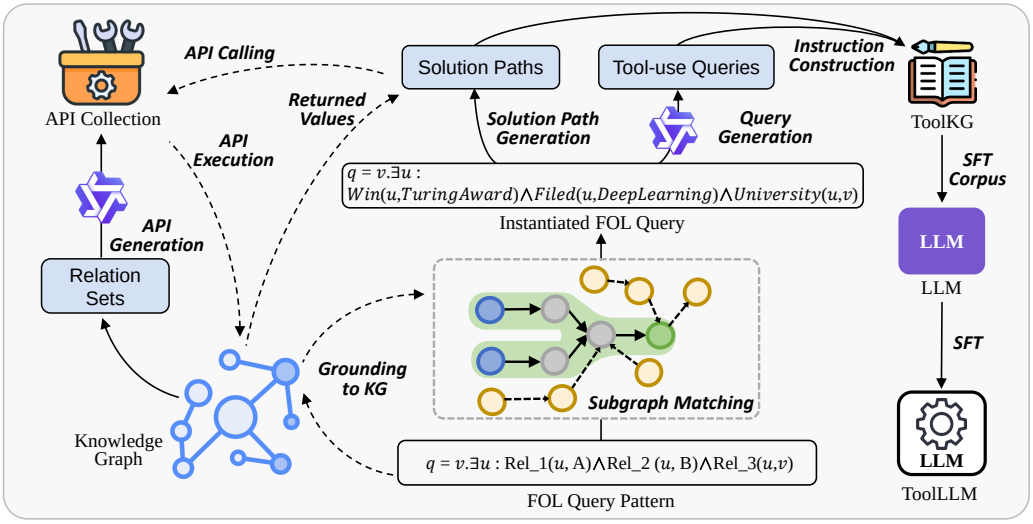


Figure 1: The framework of our proposed method. We exploit a reliable knowledge graph and introduce FOL queries as intermediates to produce tool-use queries and solution paths.

tion. Recent studies have primarily focused on the RAG framework, which retrieves relevant KG subgraphs and integrates them into the input context using well-designed prompts. For example, RoG (Luo et al., 2023) proposes a planning-retrieval-reasoning framework to generate relation paths from KGs, enabling valid reasoning for LLMs. StructGPT (Jiang et al., 2023) introduces an iterative reading-then-reasoning framework that allows LLMs to access structured knowledge through specialized interfaces, aiding answer generation. Beyond prompt-based methods, (Zhou et al., 2024b) shows that SFT with synthetic graph-based reasoning data can effectively improve LLMs' reasoning performance. (Wang et al., 2024) constructs planning data from KGs and fine-tunes LLMs to follow instructions and execute plans to obtain final answers. To our knowledge, our approach is the first to generate instruction-tuning data from KGs specifically for enhancing LLM tool utilization.

3 Preliminary

In this section, we describe the background information of KGs and FOL queries.

Knowledge Graphs. KGs organize and store real-world knowledge in a heterogeneous graph structure, representing entities as nodes and relations as edges. Given a set of entities $\mathcal{V}$ and a set of relations $\mathcal{R}$, a knowledge graph can be denoted as a tuple $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{R})$, where $\mathcal{E}$ is a set of triplets $\mathcal{E} = (h_i, r_i, t_i) \subseteq \mathcal{V} \times \mathcal{R} \times \mathcal{V}$. Each triplet represents a fact from head entity h_i to tail entity t_i with the relation type r_i.

First-order Logic Queries. To enable large-scale subgraph sampling from KGs and directly translate these subgraphs into natural language instructions, we introduce FOL queries. A FOL query is a formula composed of constants (denoted with the entity's name), variables (denoted with lowercase letters, e.g., u, v), relations (denoted with relation term, formatted as R(a, b)) and logic symbols (including $\exists$, $\wedge$, $\vee$, $\neg$). In our method, each constant or variable represents an entity from the set $\mathcal{V}$. Every relation symbol R(a, b) acts as a binary function, signifying whether a relation R exists between a pair of constants or variables. Regarding logical symbols, our considerations encompass conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$) and existential quantification ($\exists$). A bounded variable is free if it is quantified in the expression If a variable is quantified with an existential symbol, it is termed a bounded variable; otherwise, it is a free variable. For example, a natural language question, "Which universities do the Turing Award winners of deep learning work in?" can be equivalent to a FOL query as $q = v.\exists u : Win(u, TuringAward) \wedge Filed(u, DeepLearning) \wedge University(u, v)$ (Zhu et al., 2022). For an FOL query, our goal is to find the answers to the free variables that make the formula true.

Basic Operations on KGs. Corresponding to relations and logical symbols in FOL queries, we define four basic operations over entities to enable

reasoning on KG. The operations are as follows:

- **Relation Projection**: $P_q(A)$ computes the entity set of tail entities reachable by the input set of head entities through relation q. To derive head entities given tail entities (e.g. *q=v. Win(v, Turing Award)*), the projection is denoted as $P_{q^{-1}}(A)$ for inverse relation q^{-1}.
- **Intersection Operation**: The intersection operation, denoted as $A \cap B$, computes the entity set containing all entities common to both sets A and B.
- **Union Operation**: The union operation, denoted as $A \cup B$, computes the entity set containing all entities that belong to either set A or set B, or to both.
- **Complement Operation**: The complement operation, denoted as $U \setminus A$, computes the entity set containing all entities that are in set U but not in set A. U stands for the universal set in the current operation.

4 Data Construction

Drawing on the analogy between relations in triples and function operations, we abstract head nodes and their relations as input parameters and API calls. Building on this, we treat the search for missing nodes in a KG subgraph and its deduction process as the tool-use query and solution path, respectively. We introduce FOL queries as intermediates for generating these queries and solution paths. FOL queries naturally align with problem decomposition and multi-step solutions, and can be easily translated into natural language. Each FOL query corresponds to a unique subgraph pattern, enabling large-scale sampling of reasoning subgraphs. By leveraging these advantages, we minimize human effort (such as manual verification and modification) while maximizing data quality and generation efficiency. We will detail our data construction method using FOL queries, covering query generation, API generation, solution path generation, and instruction data construction. Our framework is illustrated in Figure 1.

4.1 API Generation

When written in the projection form $P_{r_i}(h_i) = t_i$, each fact triplet (h_i, r_i, t_i) in the knowledge graph can be understood as applying a relation-specific operation on the head node to yield the tail node. This closely parallels *function calling* in tool-using, where a function identified by its name is executed on the provided input parameters and returns outputs after performing its designated operations. Viewed from a translation perspective (Bordes et al., 2013), each relation instance within a triplet acts as a function that conducts a transformation from its head into its tail. Therefore, we abstract each relation type as an API in our approach, requiring head nodes as input parameters and returning tail nodes. While these functions are not directly collected from real-world API repositories, almost all correspond to genuine needs and functionalities in certain real scenarios. It should be noted that instead of simulating executions as ToolAlpaca (Tang et al., 2023), each API call can get accurate and verifiable results via truly executing queries in the knowledge graph.

To complete the API generation using relation types, we perform format conversions to align with API conventions and linguistic conversions to ensure appropriate function names for projections from head entities to tail entities. In most cases, this API generation process is straightforward by simply prefixing *"get_"* and postfixing *"_of_"* with *head type*. In the aforementioned query scenario, with "University" as the relation and persons as head entities, the API name can be easily derived as "get_university_of_person". When dealing with reverse relations like Win^{-1}, it requires generating an API with the inverse meaning of the original relation(i.e., winners). In practice, we design forward and reverse relation templates, utilizing in-context learning methods to prompt the LLM to generate APIs. As our API generation approach avoids excessive reasoning that could introduce errors or inaccuracies, a compact open-source LLM is adequate for this procedure.

Our API set encompasses relation-based APIs and three dedicated APIs for executing logical operations, including conjunction, disjunction, and negation. Specifically, *get_intersection_of*, *get_union_of* and *get_negation_of* serve as the corresponding APIs to perform intersection operation, union operation and complement operation. These APIs are implemented as functions and executed by Python interpreters.

4.2 FOL Instantiation and Query Generation

To facilitate large-scale data generation of diverse tool-usage queries with reasoning complexity, we use FOL queries as the intermediate form to pro-

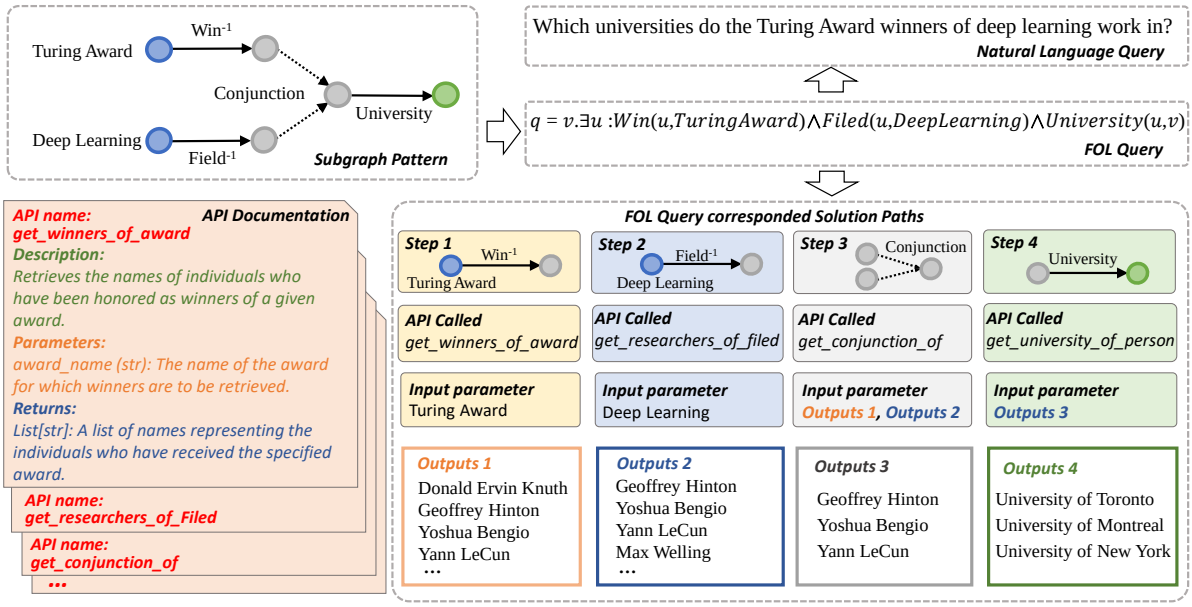


Figure 2: Demo of solution path generation from a given subgraph.

duce queries and solution paths to ensure efficient and high-quality data generation. Following prior research (Zhu et al., 2022), we define a total of 14 FOL query patterns in practice, denoted as $P = \{1p, 2p, 3p, 2i, 3i, pi, ip, 2u, up, 2in, 3in, inp, pin, pni\}$. Here, p, i, u, and n represent operations of relation projection, intersection, union, and complement. Each FOL query pattern corresponds to a specific sub-graph pattern, as shown in Appendix Table 2 and 3 with their examples.

To instantiate a FOL query pattern using its subgraph pattern, we sample real reasoning subgraphs from the KG via simple subgraph matching. Specifically, we randomly select an entity e from the KG as the root node of the tree-structured subgraph pattern. We then use a pre-order traversal to assign KG entities and relations to the subgraph structure. We uniformly select an existing relation r from e's incoming relations for each edge and assign r and its head entity e' to the corresponding edge and node in the subgraph pattern. This process continues until all nodes and relations are matched. If any required relation is missing, the instantiation fails. We use a post-order traversal for FOL subgraphs with negations to avoid cases where the complement operation results in an empty set. Once successfully instantiated, we fill the entities and relations into the FOL query pattern to obtain the instantiated FOL query. With the help of LLMs, these FOL queries can then be easily transformed into natural language tool-use queries.

Given that tool-use queries are natural language form of our obtained FOL queries, we can easily translate an FOL query (e.g., $q = v.\exists u : Win(u, TuringAward) \wedge Filed(u, DeepLearning) \wedge University(u, v)$) into the final query (e.g., "Which universities do the Turing Award winners of deep learning work in?"), while ensuring the accuracy of this procedure. We utilize a compact LLM to perform query generation from FOL quires. We present our prompt for query generation in the Appendix.

4.3 Solution Path Generation

Since FOL queries of the same pattern share identical execution sequences, we obtain the execution chains for each FOL pattern by traversing the instantiated subgraph structure in a post-order manner. With the FOL queries established, their solution paths are naturally derived by calling the APIs and recording their results. Because each API call and its response are sourced from reliable knowledge graphs, the correctness of the solution paths is ensured. For example, consider the instance in Figure 2. The FOL query dictates the decomposition steps and overall execution sequence. Initially, two APIs are called to retrieve single-step results: one for querying the winners of the *Turing Award* and another for researchers in *Deep learning*. Next, an intersection API is invoked to find overlapping researchers. Finally, a relation projection API is applied to the intermediate set to obtain the desired

answers regarding their universities.

4.4 Instruction Data Construction

At this point, we can obtain the initial query-solution pairs, where the solution includes detailed steps to resolve the query, with each step containing the required API, parameters, and the API's return results. The instruction data for fine-tuning LLMs is typically presented in a chat format. Hence, we convert the initial query-solution pairs into a dialogue format to align with standard instruction data formats (such as Alpaca or ShareGPT). Following (Chen et al., 2023, 2024), we add a system prompt to each query-solution pair, informing the assistant which tools can be called, then regard the query as the user's input and each tool invocation as output, thus constructing complete instruction-following data. Since we use high-quality knowledge graphs like FB15k (Kadlec et al., 2017) that have been manually curated, the instruction data constructed in this way does not need to be quality-checked by advanced LLMs like GPT-4, also avoiding the need for extensive manual quality control. We refer to this instruction data constructed from the knowledge graph as ToolKG and will make our code and data publicly available after the review process.

5 Evaluation and Results

5.1 Experimental Setup

Benchmark. We use the largest available tool utilization benchmark to evaluate the tool use performance of LLMs comprehensively: **T-Eval** (Chen et al., 2023). T-Eval has 23,305 test cases covering various tool sets and yields 5.8 calling steps for each query on average. T-Eval is a step-by-step tool evaluation benchmark for LLMs, which explicitly decomposes the evaluation into six sub-tasks (i.e., plan, reason, retrieve, understand, instruct, and review) along the basic capabilities of LLMs.

Training setting. To test the effectiveness of our data, we conduct extensive experiments by training LLMs with the generated ToolKG instruction data. We train the open-source LLMs, Qwen2.5-Instruct series (Team, 2024), in the SFT manner. We refer to the model trained with our data as Qwen2.5-ToolKG. For example, Qwen2.5-7B, after being fine-tuned with ToolKG, is denoted as Qwen2.5-ToolKG-7B. Due to the limited resources, we adopt the parameter-efficient training strategy LoRA (Hu et al., 2021) to fine-tune all models. As for the hyper-parameters setting, we adopt one of the most common settings, which sets the rank as 16 and alpha as 32 for all modules in the model. The learning rate is 0.0001, warmup ratio 0.1, LR scheduler $cosine$, and batch size 32. We used 2k randomly sampled data from ToolKG for instruction fine-tuning in all experiments.

Inference setting. To enhance evaluation efficiency, we adopt one of the most popular LLM inference engines, vLLM (Kwon et al., 2023), to implement the inference process of various open-source LLMs. For the inference parameters, we set the temperature to 0, top_p to 1.0, top_k to -1, and batch size to 32.

5.2 Overall Performance

Table 1 summarizes the results of various LLMs, including close-source and open-source models, and our SFT models on T-Eval. Firstly, closed-source LLMs achieve exceptionally good tool-use performance and have a significant lead compared to early open-source models. However, this gap is rapidly narrowing; for instance, the overall performance of Qwen2-7B has already far surpassed that of previous 70B-level large models (like Llama2-70B and Qwen-72B). The overall performance of Qwen2-72B (83.45) and Qwen2.5-72B (86.71) has reached or even surpassed the levels of GPT-3.5 (84.05) and GPT-4 (86.44). These results show that open-source LLMs have made significant progress in tool usage capabilities, and the gap between 7B-size and 70B-size models is also narrowing.

Further, we can see that the models fine-tuned with our ToolKG data have experienced a very significant performance improvement. For instance, Qwen2.5-ToolKG-7B improved its performance by 9.0% over Qwen2.5-7B, and its overall score (84.72) also surpassed GPT-3.5. In particular, Qwen2.5-ToolKG-14B achieved the highest score of 87.21, surpassing its 70B-scale counterpart in the same series, Qwen2.5-72B (86.71). These results demonstrate the effectiveness of our ToolKG data for tool use.

5.3 Scaling Performance of Model Size

Generally, the performance of LLMs increases with scale, as shown in Table 1. To explore whether models fine-tuned with our data exhibit a similar scale effect, we conduct experiments using the Qwen2.5-Instruct series, which offers a wide range of model sizes. Due to resource limitations, we only performed instruction fine-tuning on models

Table 1: Main Results of T-Eval. Overall stands for the score calculated from an average of metrics on all subsets. (Bold and underlined text indicate the optimal and the second-best scores, respectively.)

Model	Instruct	Plan	Reason	Retrieve	Understand	Review	Overall
GPT-3.5	96.60	86.60	67.75	**92.25**	85.50	75.60	84.05
GPT-4	96.30	87.80	65.35	88.95	85.75	**94.50**	86.44
GPT-4o	94.01	75.54	66.56	78.86	81.73	83.84	80.09
ToolAlpaca-7B	0.47	17.26	19.07	17.73	19.52	0	12.34
LLaMA2-7B	34.45	28.05	22.10	16.90	24.45	38.60	27.43
ToolACE-8B	2.15	37.15	26.23	17.03	17.9	68.79	28.21
Vicuna-7B	48.05	30.60	48.75	22.50	60.50	58.50	44.82
InternLM-7B	39.15	55.40	36.90	47.15	50.30	46.20	45.85
ChatGLM3-6B	72.05	42.70	36.15	45.25	57.75	54.80	51.45
Mistral-7B	61.65	71.05	39.15	51.80	48.95	63.20	55.97
Baichuan2-7B	73.00	52.30	41.30	51.10	59.65	61.40	56.46
Qwen-7B	61.45	64.65	45.25	62.15	61.90	61.60	59.50
LLaMA3.1-8B	81.62	69.92	66.66	76.76	73.38	82.14	75.08
GLM4-9B	90.18	77.09	68.38	77.08	74.71	64.89	75.39
Qwen2-7B	97.66	83.02	63.92	85.65	83.41	42.51	76.03
Qwen2.5-7B	93.70	74.61	66.52	87.96	75.63	67.97	77.73
LLaMA2-13B	33.35	56.90	26.45	24.65	29.40	53.00	37.29
Vicuna-13B	48.90	39.90	52.70	20.35	65.90	60.80	48.09
Baichuan2-13B	29.85	60.80	41.85	55.70	56.00	57.30	50.25
Qwen-14B	73.65	74.65	52.35	75.60	64.65	56.90	66.30
Qwen2.5-14B	98.37	87.90	69.00	84.50	77.66	74.33	81.96
LLaMA2-70B	78.95	60.55	31.10	39.55	44.80	62.80	52.96
Qwen-72B	63.05	79.25	59.45	70.90	75.30	80.30	71.38
Qwen2-72B	98.36	86.29	71.06	89.95	<u>86.87</u>	68.17	83.45
Qwen2.5-72B	**98.75**	<u>88.61</u>	72.42	90.71	80.63	<u>89.12</u>	86.71
LLaMA3.1-70B	98.25	**89.79**	69.55	91.05	**88.83**	83.78	<u>86.87</u>
Qwen2.5-ToolKG-7B	97.36	83.83	<u>75.47</u>	90.45	84.64	76.59	84.72
Qwen2.5-ToolKG-14B	<u>98.68</u>	86.68	**77.71**	<u>91.96</u>	85.28	82.96	**87.21**

within the scale range of 0.5B to 14B. The original Qwen2.5-Instruct models are denoted as raw models, and the models after instruction fine-tuning with our data are marked as SFT models.

Both raw and SFT models are evaluated on the T-Eval benchmark, with results in Figure 3. The performance of both the raw and SFT models continues to improve as the model size increases. At the same time, the advantage of the SFT models over the raw models remains consistent, indicating that our data has the potential to enhance the tool usage capabilities of larger LLMs. Notably, the 3B model for mobile applications scored over 80 after SFT, achieving performance comparable to GPT-4o (80.09). This result suggests that small language models on the mobile side can also achieve tool use performance comparable to advanced LLMs like GPT-4o. This finding strongly supports the development of powerful LLM applications for mobile devices.

5.4 Study on Various Backbone LLMs

To verify whether our data is also effective for other LLMs, we selected two other models that are similar in size to Qwen2.5-7B and widely followed by the community (i.e., Llama3.1-8B-Instruct and GLM4-9B-Chat (GLM et al., 2024)) for experimentation. The performance of these models before and after fine-tuning on T-Eval is displayed in Figure 4.

It can be seen that for different types of LLMs, our ToolKG data is always effective, and the performance improvements of the SFT models are sig-

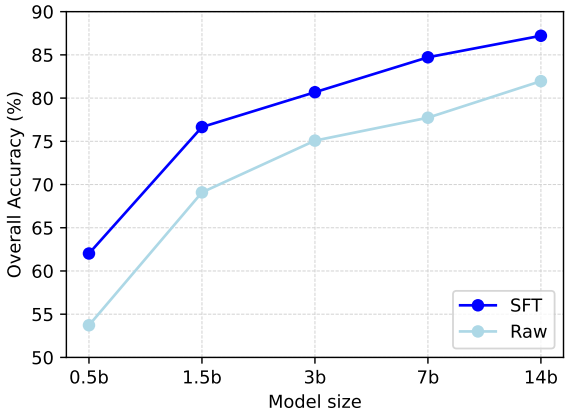


Figure 3: Performance scaling laws for the parameters of training models, from 0.5B to 14B.

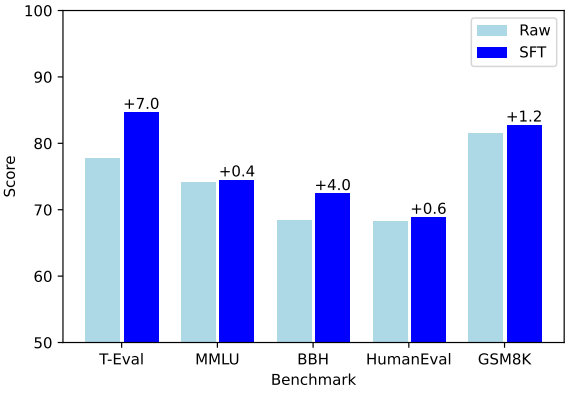


Figure 5: General performance of Qwen2.5-7B-Instruct fine-tuned with ToolKG.

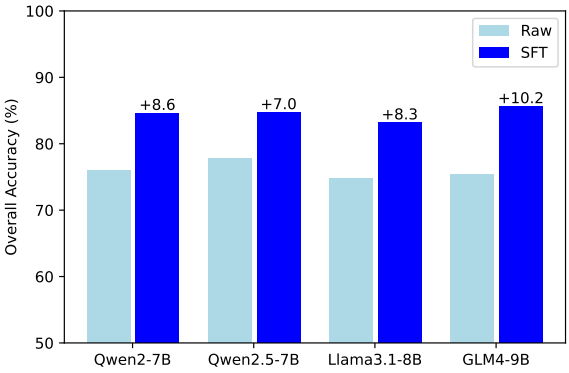


Figure 4: Performance of various backbone LLMs fine-tuned with ToolKG.

nificant. Among the raw models, the performance of Llama3.1-8B and GLM4-9B is inferior to that of Qwen2-7B and Qwen2.5-7B, which may be due to differences in model architecture. However, after fine-tuning with our data, their performance gains are very significant, especially for GLM4-9B, which improved from 75.4 to 85.6, surpassing Qwen2.5-ToolKG-7B. This result demonstrates that smaller language models below the 10B size can also perform excellent tool utilization comparable to LLMs like GPT-4.

5.5 Study on General Capabilities

The results above indicate that LLMs fine-tuned with our constructed instruction data possess superior tool usage capabilities. However, whether their abilities in other aspects are affected also needs further evaluation. To assess the impact of SFT with ToolKG on the broader capabilities of LLMs, we conduct experiments using Qwen-2.5-7B-Instruct across various benchmarks evaluating general ability (MMLU (Hendrycks et al., 2021), BBH (Suzgun et al., 2023)), coding (HumanEval (Chen et al., 2021)), mathematics (GSM8K (Cobbe et al., 2021)), and tool utilization (T-Eval).

Figure 5 presents the performance metrics for both raw and SFT models across these benchmark evaluations. It can be seen from the figure that the SFT model outperforms the RAW model in all benchmarks, indicating that fine-tuning with our ToolKG data not only fails to weaken the performance of the original model but can even have a positive impact. Notably, the SFT model scored 4.0 points higher than the Raw model on the BBH benchmark, indicating that our data is also significantly effective for the BBH task. Since the BBH task requires multi-step reasoning, it aligns well with how we construct our ToolKG data. This result suggests that our data can potentially improve the complex reasoning abilities of LLMs.

6 Conclusion

This paper proposes a novel data synthesis method that utilizes knowledge graphs to generate high-quality instruction data to enhance the tool utilization performance of LLMs. By leveraging the structural and semantic information of knowledge graphs, we generated API tool sets, queries, and their corresponding solution path sets, thereby constructing the instruction dataset ToolKG. Through extensive experiments, we demonstrate that smaller language models fine-tuned with our ToolKG data can achieve state-of-the-art tool utilization performance surpassing GPT-4 while maintaining good general capabilities. Our results show that the high quality of data synthesized from KGs can potentially enhance the general capabilities of LLMs.

Limitations

We have demonstrated that synthesizing a small amount of data using knowledge graphs can significantly enhance the tool-use capabilities of LLMs. The used knowledge graph should contain many diverse entities and relationships. Small-scale knowledge graphs with limited entities and relationships may not achieve such significant effects.

References

Stephen H Bach, Victor Sanh, Zheng-Xin Yong, Albert Webson, Colin Raffel, Nihal V Nayak, Abheesht Sharma, Taewoon Kim, M Saiful Bari, Thibault Fevry, et al. 2022. Promptsource: An integrated development environment and repository for natural language prompts. *arXiv preprint arXiv:2202.01279*.

Antoine Bordes, Nicolas Usunier, Alberto Garcia-Duran, Jason Weston, and Oksana Yakhnenko. 2013. Translating embeddings for modeling multi-relational data. *Advances in neural information processing systems*, 26.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Zehui Chen, Weihua Du, Wenwei Zhang, Kuikun Liu, Jiangning Liu, Miao Zheng, Jingming Zhuo, Songyang Zhang, Dahua Lin, Kai Chen, et al. 2023. T-eval: Evaluating the tool utilization capability step by step. *arXiv preprint arXiv:2312.14033*.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. 2024. Agent-flan: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. In *International Conference on Machine Learning*, pages 10764–10799. PMLR.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2021. Measuring massive multitask language understanding. In *International Conference on Learning Representations*.

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*.

Jinhao Jiang, Kun Zhou, Zican Dong, Keming Ye, Wayne Xin Zhao, and Ji-Rong Wen. 2023. Structgpt: A general framework for large language model to reason over structured data. *arXiv preprint arXiv:2305.09645*.

Rudolf Kadlec, Ondrej Bajgar, and Jan Kleindienst. 2017. Knowledge base completion: Baselines strike back. *arXiv preprint arXiv:1705.10744*.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *arXiv preprint arXiv:2212.14024*.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. 2024. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and

Jianfeng Gao. 2024. Chameleon: Plug-and-play compositional reasoning with large language models. *Advances in Neural Information Processing Systems*, 36.

Linhao Luo, Yuan-Fang Li, Gholamreza Haffari, and Shirui Pan. 2023. Reasoning on graphs: Faithful and interpretable large language model reasoning. *arXiv preprint arXiv:2310.01061*.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2024. Tool learning with foundation models. *Preprint*, arXiv:2304.08354.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2024. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2024. Hugginggpt: Solving ai tasks with chatgpt and its friends in hugging face. *Advances in Neural Information Processing Systems*, 36.

Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. 2023. Nexusraven: a commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, et al. 2023. Challenging big-bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 13003–13051.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Junjie Wang, Mingyang Chen, Binbin Hu, Dan Yang, Ziqi Liu, Yue Shen, Peng Wei, Zhiqiang Zhang, Jinjie Gu, Jun Zhou, et al. 2024. Learning to plan for retrieval-augmented large language models from knowledge graphs. *arXiv preprint arXiv:2406.14282*.

Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R Fung, Hao Peng, and Heng Ji. 2023. Craft: Customizing llms by creating and retrieving from specialized toolsets. *arXiv preprint arXiv:2309.17428*.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2024a. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36.

Jiaming Zhou, Abbas Ghaddar, Ge Zhang, Liheng Ma, Yaochen Hu, Soumyasundar Pal, Mark Coates, Bin Wang, Yingxue Zhang, and Jianye Hao. 2024b. Enhancing logical reasoning in large language models through graph-based synthetic data. *arXiv preprint arXiv:2409.12437*.

Zhaocheng Zhu, Mikhail Galkin, Zuobai Zhang, and Jian Tang. 2022. Neural-symbolic models for logical queries on knowledge graphs. In *International conference on machine learning*, pages 27454–27478. PMLR.

A Appendix

A.1 FOL Query Patterns

Tables 2 and 3 depict all the 14 FOL query patterns and their corresponding FOL Subgraphs in detail. In each row, we provide an instantiated query example in FOL form and its translated query in natural language form for each FOL query pattern.

Table 2: Illustration of FOL Query Patterns and FOL Subgraphs with Query Examples in FOL and Natural Language Form (Part 1)

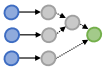
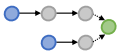
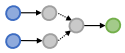
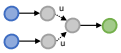
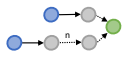
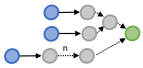
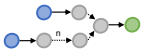
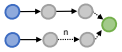
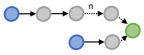
Query Type	FOL Subgraph	FOL Query Pattern	Instantiated Query Example in FOL Form	Query Example in Natural Language Form
1p		q =?a : Rel_1(A, a)	q =?a : Star(Amy Irving, a)	What film did Amy Irving star in?
2p		q =?b : Rel_1(A, a) ∧ Rel_2(a, b)	q =?b : Nominated(Lee Grant, a) ∧ Winner(a, b)	Who is the winner of the award that Lee Grant was nominated for?
3p		q =?c : ∃ a, b : Rel_1(A, a) ∧ Rel_2(a, b) ∧ Rel_3(b, c)	q =?c : ∃ a, b : MemberStates(WTO, a) ∧ JurisdicationOfOffice(b, a) ∧ Organization(b, c)	What is the organization that a politician of a WTO member state came from?
2i		q =?c : Rel_1(A, c) ∧ Rel_2(B, c)	q =?c : Genre(c, Science Fiction) ∧ DistributeFilm(Warner Bros., c)	Which science fiction film is distributed by Warner Bros.?
3i		q =?e : Rel_1(A, e) ∧ Rel_2(B, e) ∧ Rel_3(C, e)	q =?e : Profession(e, songwriter) ∧ WinnerOfSameAward(e, BeBe Winans) ∧ NominatedFor-SameAward(Babyface, e)	Which songwriter won the same award as BeBeWinans, and was nominated for the same award as Babyface?
pi		q =?d : ∃ a : Rel_1(A, a) ∧ Rel_2(a, d) ∧ Rel_3(B, d)	q =?d : ∃ a : Student(West Point, a) ∧ Found(a, d) ∧ Company(Buzz Aldrin, d)	What is contained by both the administrative division of Columbia and South Carolina?
ip		q =?d : ∃ c : Rel_1(A, c) ∧ Rel_2(B, c) ∧ Rel_3(c, d)	q =?d : ∃ c : Award(Freddy Got Fingered, c) ∧ Nominated(Peter Hyams, c) ∧ NominatedFor(d, c)	What film is nominated for the award that Freddy Got Fingered won and Peter Hyams was nominated for?
2u		q =?c : Rel_1(A, c) ∨ Rel_2(B, c)	q =?c : Student(Bucknell University, c) ∨ Nominated(c, National Book Award for Fiction)	What genre is played by Chick Corea or Keith Jarrett?
up		q =?d : ∃ c : (Rel_1(A, c) ∨ Rel_2(B, c)) ∧ Rel_3(c, d)	q =?d : ∃ c : (Award(David Kirschner, c) ∨ Ceremony(c, 39th Daytime Emmy Awards)) ∧ Award(d, c)	Who wins the award that David Kirschner winned or is given at 39th Daytime Emmy Awards?

Table 3: Illustration of FOL Query Patterns and FOL Subgraphs with Query Examples in FOL and Natural Language Form (Part 2)

Query Type	FOL Subgraph	FOL Query Pattern	Instantiated Query Example in FOL Form	Query Example in Natural Language Form
2in		q =?d : Rel_1(A, d) ∧ ¬Rel_2(B, d)	q =?d : MortgageSource(d, US Department of HUD) ∧ ¬PlaceLive(Allison Janney, d)	Which city takes mortgage from US Department of Housing and Urban Development, but Allison Janney hasn't lived in?
3in		q =?f : Rel_1(A, f) ∧ Rel_2(B, f) ∧ ¬Rel_3(C, f)	q =?f : ReleaseMedium(f, DVD) ∧ ReleaseRegion(f, New Zealand) ∧ ¬Language(f, English)	Which film has a DVD version and is released in New Zealand, but doesn't have an English version?
inp		q =?e : ∃ d : Rel_1(A, d) ∧ ¬Rel_2(B, d) ∧ Rel_1(d, e)	q =?e : ∃ d : FieldOfStudy(McGill University, d) ∧ ¬Language(Nico, d) ∧ FieldOfStudy(e, d)	Who studies the field that is studied by McGill University, but is not spoken by Nico?
pin		q =?e : ∃ a : Rel_1(A, a) ∧ Rel_2(a, e) ∧ ¬Rel_3(B, e)	q =?e : ∃ a : Country(a, USA) ∧ Student(a, e) ∧ ¬Film(e, Malcolm X)	Who was a student of a university in United States, but did not film Malcolm X?
pni		q =?e : ∃ a : Rel_1(A, a) ∧ ¬Rel_2(a, e) ∧ Rel_3(B, e)	q =?e : ∃ a : SymptomOf(dyspnea, a) ∧ ¬CauseOfDeath(e, a) ∧ Religion(e, Catholicism)	Who believed in Catholicism and did not die from the disease that has the symptom of dyspnea?

A.2 Prompt Content

Figures 6 and 7 show the prompt for API generation using triple relations in knowledge graphs. Tables 8, 9 and 10 show the prompt to translate an instantiated FOL query of a specific pattern into its natural language form. We use a single prompt to convert FOL queries of all patterns into natural language questions. This prompt includes typical examples for each pattern for the LLM to reference. It is worth noting that, to avoid redundancy, only the conversion examples for *1p* and *ip* patterns are shown here. The detailed prompt can be accessed in our later-released project repository after review.

Toolset Construction Prompt

```
# Role
You are a senior programmer responsible for writing APIs based on my
instructions and examples.

## API Generation Instructions
I will provide the following relations as input, along with corresponding
examples of triples (head, relation, tail). You need to convert the relation
into an API format so that after passing the head as an input parameter and
executing the API converted by the relation, you obtain the tail.
Note that you need to observe and summarize the functionality of the triples to
ensure they can be executed successfully by passing in the head to obtain the
tail.

### Relation Representation
The relation in triples (head, relation, tail) can be either a basic relation
or a composite relation.

#### Basic Relation Representation
The basic representation of a single relation in the form of
`/film/film/written_by` includes:
- The topic domain: `film`
- The head type: `film`
- The tail type: `written_by`

#### Composite Relation Representation
Some relations are composite, formed by two basic relations connected by a '.',
with the intermediate entity omitted.
`/film/film/distributors./film/film_film_distributor_relationship/region`, for
example, indicates the relationship from head to mid combined with mid to tail,
where the mid entity is omitted to form a composite relation. Therefore, by
omitting the intermediate, this relation represents the relationship from a
film to its distributed region.

## Task
Your task is to generate API documentation based on the provided relation.
You need to think thoroughly and understand the function of the relation that
transforms head to tail based on the above Relation Representation.

Please generate API documentation that conforms to a specified JSON format.
This documentation should clearly define the API name, function description,
input parameter properties, required parameters, and additional properties.
Pay careful attention to the structure and formatting as outlined in the
examples below.

### API Naming Conventions
1. **API Naming Format**:
- Form the API name by combining the domain category and the function name,
following the format: 'domain.function_name'.
2. **Domain Classification**:
- Classify each API into a specific domains, such as "education", "film",
"sports", etc.
- If a common topic arises, classify it under an additional "common" category.
```

Figure 6: Prompt for API Generation using Relations (Part1).

Toolset Construction Prompt

````
## Example Inputs and Expected Outputs
Given the input example:
```json
{
    "relations":
"/award/award_nominee/award_nominations./award/award_nomination/award_nomin
ee",
    "triples": [
        [
            "Danny_DeVito",

"/award/award_nominee/award_nominations./award/award_nomination/award_nomin
ee",
            "Guy_Pearce"
        ]
    ]
}
```
Expected Output:
```json
{
    "name": "award.get_award_nominees_of_the_same_award",
    "description": "Get the nominees who were nominated alongside the given
award nominee",
    "parameters": {
        "type": "object",
        "properties": {
            "award_nominee": {
                "type": "string",
                "description": "The name of the award nominee"
            }
        },
        "required": [
            "award_nominee"
        ],
        "additionalProperties": false
    }
}
```

## Let's get started!
Given input:
```json
__INPUT_JSON__
```

Please analyze the given input carefully and provide the output after
"Expected Output:". Attention!! The output should only present results
conforming to the required JSON format, and it is forbidden to give
reasoning processes or other explanations.
````

Figure 7: Prompt for API Generation using Relations (Part2).

Query Generation Prompt

```
# Role
You are a translator responsible for converting First-order Logical (FOL)
queries into natural language questions based on the provided examples and
guidelines.

## Translation Instructions from FOL query to natural language question
I will provide you with a First-order Logical (FOL) query, and your task is to
translate it into a clear and unambiguous natural language question.

An FOL query is structured in the format like `q =?b: pred1(A,a) ∧ pred2(B,a)
∧ pred3(a,b)`, where `A` and `B` are constants that will be instantiated by
specific entities, `a` is an existential variable, and `pred1`, `pred2`,
`pred3` will be replaced by specific predicate names.

Following `predicate_domain.get_predefined_property_of_subject(subject,
object)` format, each predicate retrieves a specific property of the subject to
yield objects. E.g., the predicate `film.get_language_of_film("The Tourist",
"Russian Language")` indicates that the language of the film "The Tourist" is
"Russian Language". To help you understand, each predicate name is accompanied
by its corresponding predicate description and a subject description.

The queried value is the variable that follows the question mark `?`, such as
`b` in `?b`. In the FOL pattern `q =?b: pred1(A,a) ∧ pred2(B,a) ∧ pred3(a,b)`,
the expression is used to find the variable `b`, such that there exists a
variable `a` for which the predicate `pred1` applied to constant `A` and the
predicate `pred2` applied to constant `B` both result in `a`. For every `a`
meeting these conditions, the goal is to query the objects of predicate `pred3`
using each `a` as the subject.

There are several FOL patterns. The FOL structure mentioned earlier is one of
these patterns. I will provide examples for each pattern. You need to
understand these examples and complete the translation accordingly.

**Instructions for Generating Questions:**
1. Ensure the generated question is grammatically correct and follows standard
English conventions.
2. Use clear and concise language to make the question easy to understand. Keep
the question straightforward and coherent.
3. Avoid complex structures; but ensure not to omit any steps or critical
information. Use multiple clauses for longer questions to enhance clarity and
flow.
4. Write in a style that aligns with natural human reading habits, ensuring the
question is both accurate and fluent.
5. Your question must be clear and unambiguous, especially for reference.
6. Please avoid directly copying text from the predicate name and description
into the questions. Instead, only use them to understand the predicate meaning.
Rephrase the predicates using more concise and natural language to express the
concepts where possible.
7. Ensure that you follow the logical flow of the questions in a multi-step
question. Pay special attention to maintaining coherence and alignment in your
translated question when the object of a predicate is used as the subject for
the next step.
```

Figure 8: Prompt for Translating Instantiated FOL Queries into Natural Language Form (Part1).

Query Generation Prompt

````
## Example Inputs and Expected Outputs

### Examples in FOL_pattern: 1p
Given the FOL Query example:
```json
{
    "FOL_pattern": "q =?a: pred1(A,a)",
    "FOL_query": "q =?a: people.get_profession_of_person(Fred Willard,a)",
    "predicates": [
        {
            "predicate_name": "people.get_profession_of_person",
            "predicate_description": "Get the profession of the given person.",
            "subject": {
                "person": "The name of the person."
            }
        }
    ]
}
```
Expected Output:
```json
{
    "Natural Language Question": "What is the profession of Fred Willard?"
}
```

### Examples in FOL_pattern: ip
Given the FOL Query example:
```json
{
    "FOL_pattern": "q =?b: pred1(A,a) ∧ pred2(B,a) ∧ pred3(a,b)",
    "FOL_query": "q =?b: people.get_places_lived_of_person(Tanikella Bharani,a)
∧ sports.get_countries_winning_olympic_medal(Silver medal,a) ∧
people.get_persons_of_nationality(a,b)",
    "predicates": [
        {
            "predicate_name": "people.get_places_lived_of_person",
            "predicate_description": "Get the locations where the given person
has lived",
            "subject": {
                "person": "The name of the person"
            }
        },
        {
            "predicate_name": "sports.get_countries_winning_olympic_medal",
            "predicate_description": "Get the countries that have won a
specific Olympic medal",
            "subject": {
                "medal": "The type of Olympic medal (e.g., Bronze, Silver,
Gold)"
            }
        },
````

Figure 9: Prompt for Translating Instantiated FOL Queries into Natural Language Form (Part2).
````

Query Generation Prompt

```
        {
            "predicate_name": "people.get_persons_of_nationality",
            "predicate_description": "Get the persons who are nationals of the
given country",
            "subject": {
                "nationality": "The name of the country"
            }
        }
    ]
}
```
Expected Output:
```json
{
    "Natural Language Question": "Whose country has been inhabited by Tanikella
Bharani and also has won silver medals at the Olympics?"
}
```

Examples in other FOL_pattern
Given the FOL Query example:
…
Expected Output:
…

Let's get started!

Given the FOL Query:
```json
__INPUT_JSON__
```
Please analyze the given input carefully and provide the output after "Expected
Output:". Attention!! The output should only present results conforming to the
required JSON format. You have only one chance to give the JSON output and
Post-hoc explanations or refinements in ANY Form are NOT allowed!!

Figure 10: Prompt for Translating Instantiated FOL Queries into Natural Language Form (Part3).