Code-on-Graph: Iterative Programmatic Reasoning via Large Language Models on Knowledge Graphs

Anonymous ACL submission

Abstract

Knowledge Graphs (KGs) have emerged as a powerful tool for mitigating the inherent limitations of Large Language Models (LLMs), such as outdated knowledge and hallucinations. Existing LLM–KG integration frameworks mainly adopt a fixed set of pre-defined operators to interact with KG, subsequently injecting the retrieved factual knowledge directly into prompts to obtain the answer. This paradigm faces two critical bottlenecks: 1) **Inflexibility**: Predefined operators restrict generalization to the diverse reasoning patterns required by open-domain questions. 2) **Unscalability**: Direct injection of factual knowledge into prompts limits scalability and reliability in handling large-scale factual knowledge. To address these two bottlenecks, we propose Code-on-Graph (CoG), a programmatic reasoning framework that compiles massive factual knowledge into schema-level Python classes and enables flexible reasoning through schema-level code generation. Specifically, CoG iteratively decomposes the question into subtasks and gradually reaches the final answer. For each subtask, CoG first maps the schemas of relevant factual knowledge into Python classes. Then, CoG generates and executes the code according to these classes and obtains the results. Experiments on WebQSP, CWQ, and GrailQA demonstrate that CoG outperforms prior state-of-the-art models by up to **10.5**% and achieves up to a **1189**× improvement in Token Utility Rate.

1 Introduction

Large Language Models (LLMs)(Zhao et al., 2025; Cahyawijaya et al., 2024) have demonstrated impressive Natural Language Processing (NLP) capabilities, significantly enhancing various tasks such as question answering (Wang et al., 2023; Li et al., 2023b), text generation (Shi et al., 2025), and textual reasoning (Wei et al., 2023). Despite their success, LLMs remain susceptible to out-of-date knowledge and hallucinations, indicating that substantial challenges persist. Given that KGs(Auer et al., 2007; Bollacker et al., 2008), as large-scale and high-quality knowledge bases, can provide accurate and up-to-date information to LLMs, many studies (Sun et al., 2024; Chen et al., 2024; Li et al., 2024a; Jiang et al., 2024) have attempted to integrate LLMs with KGs to mitigate the aforementioned issues.

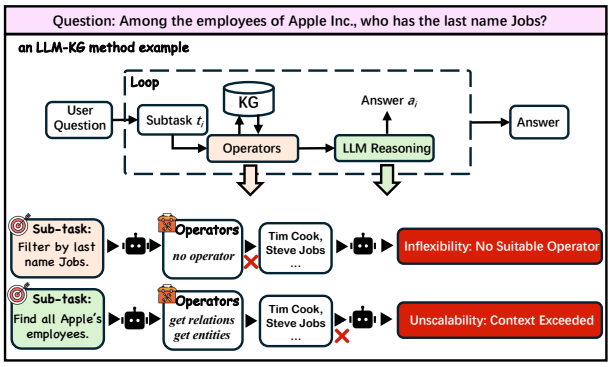

Figure 1: Limitations of the existing agent-based methods.

The dominant LLM–KG integration methods equip LLMs with external retrievers to search for question-relevant facts, which are then injected into the prompt for answer generation. Recently, LLM-KG integration methods have achieved promising performance. As illustrated in Figure 1, these methods decompose questions into subtasks and interact with KG via predefined tools to retrieve relevant factual knowledge (e.g., entities and relations). Based on this information, the LLM conducts reasoning to obtain results for each subtask to finally produce the final answer. However, these methods face two critical bottlenecks: 1) **Inflexibility**: They rely on a fixed predefined set of operators, such as "retrieval", "get_entities", and "get_relations", which inherently limits their generalization to the diverse reasoning patterns required by open-domain questions. For example, for the

query "Among the employees of Apple Inc., who has the last name Jobs?", it is hard to flexibly filter irrelevant factual knowledge via limited predefined tools. 2) **Unscalability**: Although predefined operators partially reduce the scale of retrieved factual knowledge, directly injecting the retrieved factual knowledge into prompts limits scalability and reliability when handling large-scale knowledge(Luo et al., 2024b). As the amount of relevant facts grows, prompt-based injection becomes increasingly inefficient, error-prone, and difficult to control, leading to degraded reasoning performance.

Actually, KG schemas provide a highly abstract representation of large collections of factual knowledge, offering an effective way to guide LLMs to operate over large-scale facts. The key challenge, however, lies in enabling LLMs to understand complex schemas and adaptively create task-specific tools conditioned on the input question. Notably, in programming languages (e.g., Python), schemas can be naturally expressed as classes. Together with the flexible and expressive function composition ability, code is able to represent flexible operations over large-scale factual knowledge in a concise manner.

We instantiate these ideas in Code-on-Graph (CoG), an iterative framework that consists of three stages: (1) Planning, which decomposes a complex question into subtasks; (2) Coding, which retrieves subgraphs, abstracts the retrieved subgraphs into class definitions using schema constraints, and generates operations (such as sorting, arithmetic calculations, selection, etc.); and (3) Executing, which executes the operations in a sandbox environment by combining the class definitions with the instantiation code, to obtain subtask results; if execution fails, use the error feedback for correction and retry. CoG thus enables effective and traceable reasoning over large-scale factual knowledge. Our main contributions are as follows:

- We propose CoG, an iterative programmatic reasoning framework that answers complex questions through a planning–coding–executing loop, where the model plans subtasks, generates executable code, and iteratively executes and debugs it to obtain the final answer.
- By representing schemas with Python classes and conducting schema-level code generation and execution, CoG enables flexible operations over massive factual knowledge with minimal token usage, with up to **1189**× improvement in Token Utility Rate compared to prior state-of-the-art methods on CWQ.
- By improving both the scalability and the flexibility, CoG effectively handles complex queries and outperforms previous state-of-the-art methods by up to **10.5**%.

2 Related Work

2.1 LLM-KG Integration

Early integration of LLMs with KGs primarily followed the semantic parsing paradigm, where models translate natural language into logical forms like SPARQL or Cypher to query the database (Luo et al., 2024a; Li et al., 2023a). While effective on closed schemas, these methods suffer from brittleness when facing schema inconsistencies or complex multi-hop reasoning requirements. To improve robustness, recent research has shifted toward agentic reasoning frameworks. Representative methods such as Think-on-Graph (ToG) (Sun et al., 2024) and Plan-on-Graph (PoG) (Chen et al., 2024) treat the LLM as an autonomous agent that iteratively explores the graph structure. Similarly, frameworks like StructGPT (Jiang et al., 2023a), KG-Agent (Jiang et al., 2024), and RoG (Luo et al., 2024b) utilize iterative tool invocation or retrieval-augmented generation (RAG) (He et al., 2024) to navigate external knowledge. However, these methods largely operate at the instance level—traversing individual triple node-by-node. This granularity often leads to search space explosion and noise accumulation, as agents struggle to maintain a high-level structural overview.

2.2 Program-Aided Reasoning

This paradigm demonstrates that decoupling reasoning (via program synthesis) from computation (via execution) significantly mitigates hallucination, particularly in mathematical and algorithmic tasks (Gao et al., 2023; Chen et al., 2023). This success has extended to general tool-augmented agents (Schick et al., 2023; Lu et al., 2023) and neuro-symbolic methods for structured data, such as Binder (Cheng et al., 2023) for tables and ViperGPT (Surís et al., 2023) for visual reasoning. Despite these advances, application in the KGQA (Jiang et al., 2023b) domain remains limited. Existing program-aided KG approaches typically rely on generating standard query languages or invoking

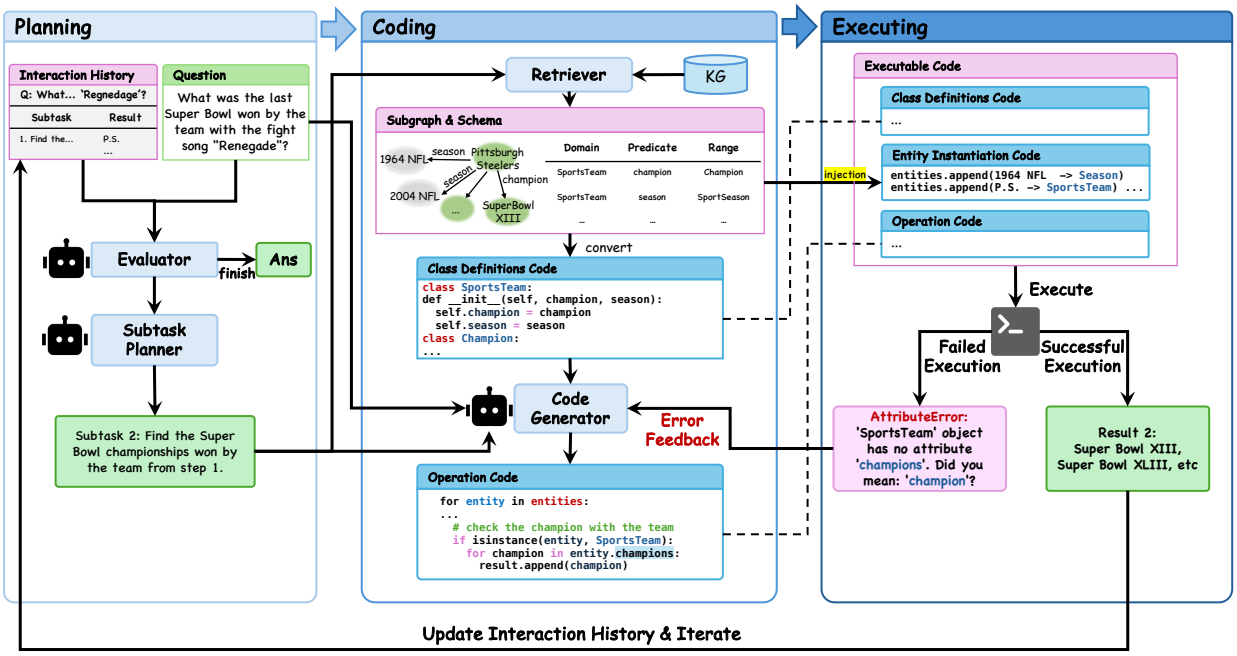

Figure 2: The framework of CoG.

a static set of pre-defined function primitives (e.g., rigid hop or filter operators). These fixed operators lack the flexibility to handle the diverse reasoning patterns found in open-domain questions. Inspired by object-oriented programming, our CoG framework advances this field by synthesizing dynamic Python code over abstracted KG objects. This allows the model to "write its own tools" on the fly, offering superior generalization compared to rigid API calls.

3 Preliminaries

Follow the previous works (Sun et al., 2024; Chen et al., 2024), in this paper, we evaluate CoG on KGQA, a representative and widely studied task for LLM–KG integration that aims to answer natural language questions through reasoning over KGs.

3.1 Knowledge Graph

A KG is a structured representation of factual knowledge, typically modeled as a set of relational triples:

$$\mathcal{G} = \{(e, r, e') \mid e, e' \in \mathcal{E},\ r \in \mathcal{R}\}, \qquad (1)$$

where $\mathcal{E}$ denotes the set of entities and $\mathcal{R}$ denotes the set of relations. Each triple (e, r, e') represents a factual assertion that relation r holds between a head entity e and a tail entity e'.

3.2 KGQA

Formally, given a natural language question q, a KG $\mathcal{G}$, and a set of topic entities $\mathcal{T}_q \subseteq \mathcal{E}$ mentioned in q, the goal of KGQA is to predict a set of answers $\mathcal{A}_q$ that correctly answer the question.

Following common practice in prior work, we assume that topic entities and answer entities are already linked to their corresponding KG identifiers:

$$\mathcal{T}_q, \mathcal{A}_q \subseteq \mathcal{E}. \qquad (2)$$

This setting isolates the reasoning component of KGQA by bypassing entity linking, allowing the model to focus on semantic understanding and multi-hop inference over the graph.

In many benchmark datasets, each question is associated with a gold reasoning path or an equivalent structured query (e.g., SPARQL), which defines a sequence of relations connecting topic entities to answer entities. However, during inference, such gold paths are not available and must be implicitly or explicitly inferred by the model.

4 Methodology

CoG is a programmatic reasoning framework, which iteratively decomposes complex questions into subtasks and performs schema-level code reasoning over a KG. As illustrated in Figure 2, each CoG iteration comprises three core modules: (1) **Planning**, which decomposes the question into manageable subtasks to progressively reach the

answer; (2) **Coding**, which maps the schemas of retrieved subgraphs into Python classes and generates code based on these classes; and (3) **Executing**, which executes and refines the code over instantiated classes in a loop to obtain the final results.

4.1 Planning

To handle complex multi-hop questions, the planning module dynamically decomposes the original query into a sequence of subtasks. As illustrated in Figure 2, the planning module comprises two core components: a *Subtask Generation*, which proposes the next subtask, and an *Evaluation*, which decides whether the planning process should terminate.

Evaluation. To avoid the inflexibility and redundant reasoning paths inherent in static decomposition, the generator $\mathcal{M}_{gen}$ dynamically determines the next subtask by leveraging the full history of prior reasoning. By conditioning on past subtasks and their execution results, the generator avoids unproductive directions and enables non-linear exploration, allowing the current subtask to build upon any relevant previous step rather than strictly following the immediate predecessor.

Formally, we represent the execution history up to step $t-1$ as $\mathcal{E}_{t-1} = [(T_1, R_1), \ldots, (T_{t-1}, R_{t-1})]$, where each pair consists of a subtask T_i and its corresponding execution result R_i. The generator then produces the next subtask T_t as:

$$T_t = \mathcal{M}_{gen}(Q, \mathcal{E}_{t-1}). \tag{3}$$

Each generated subtask T_t is a structured representation that includes a natural-language description, along with associated topic entities and candidate predicates for retrieval, which facilitate subsequent code generation. Notably, each subtask maintains a parent step reference that specifies the earlier step on which it builds, enabling CoG to capture complex, non-sequential dependencies beyond a strictly linear execution order.

Subtask Generation. Following the initial generation and execution of the first subtask, the evaluator $\mathcal{M}_{eval}$ determines at each subsequent iteration $t > 1$ whether further exploration is required. By assessing the previously proposed subtasks $\mathcal{T}_{t-1} = [T_1, \ldots, T_{t-1}]$ and the latest execution result R_{t-1}, the evaluator enables adaptive termination from two complementary perspectives: (1) **Exploration Sufficiency**: whether crucial subtasks decomposed from the question (e.g., satisfy specific constraints such as temporal qualifiers or superlatives) have been fully addressed; (2) **Answer Plausibility**: whether the most recent result R_{t-1} is type-consistent and semantically aligned with the expected answer form (e.g., entity, numeric value, or date).

Formally, at iteration $t > 1$, the evaluator predicts a binary continuation decision $c_t \in \{\texttt{True}, \texttt{False}\}$ as:

$$c_t = \mathcal{M}_{eval}(Q, \mathcal{T}_{t-1}, R_{t-1}). \tag{4}$$

If $c_t = \texttt{True}$, the process triggers the *Generator* to produce T_t; otherwise, the planning process terminates and proceeds to summarize the final answer.

4.2 Coding

To resolve each subtask T_t, the coding module follows three sequential steps: **subgraph retrieval**, **schema-class mapping**, and **code generation**. By compiling large-scale factual knowledge into concise schema-level Python classes, this module enables flexible reasoning through code generation. First, it extracts a relevant subgraph from the KG using the topic entities and candidate predicates associated with the subtask. Next, rather than handling raw triples, the module abstracts the retrieved schemas into Python class definitions. Finally, it generates executable code that instantiates these classes and performs logical operations to resolve the subtask.

Subgraph Retrieval. Since reasoning over the entire knowledge graph $\mathcal{G}$ is computationally expensive and prone to noise, this submodule retrieves a compact subgraph $\mathcal{G}_{sub} \subset \mathcal{G}$ based on the topic entities and predicates identified in the subtask T_t. Specifically, beginning from the anchor entities, the module performs a bounded expansion with a maximum depth d. To control the branching factor, it considers both outgoing and incoming edges for each visited node, retaining only the top-K edges per direction. This neighborhood selection is guided by the semantic similarity between candidate relations and the combined context of the current subtask and its target predicates. An off-the-shelf, pre-trained DistilBERT model (Sanh et al., 2020)[1] is employed to compute these similar-

[1] https://huggingface.co/sentence-transformers/msmarco-distilbert-base-tas-b

ity scores. The resulting $\mathcal{G}_{sub}$ serves as a dense yet focused evidence set for the subsequent abstraction and execution stages.

Schema-to-Class Mapping. Linearizing retrieved triples into plain text often obscures the inherent graph structure and consumes excessive context window space with redundant information. To address the unscalability and limited flexibility caused by such direct fact injection, the coding module abstracts the factual knowledge in $\mathcal{G}_{sub}$ into an executable object-oriented representation by converting the subgraph into a concise set of Python classes.

To enable LLMs to comprehend complex schemas and adaptively programming to the input question, in this step, CoG leverages the natural correspondence between KG schemas and object-oriented programming, where schemas are expressed as Python classes. By transforming $\mathcal{G}_{sub}$ into schema-level Python classes, the module overcomes the limitations of raw triple linearization, thereby providing a more scalable and flexible coding interface.

The mapping process is performed in two stages. First, for each predicate p within the retrieved subgraph, the module utilizes schema constraints to infer the class types for the head and tail entities of each triple (h, p, t). Subsequently, each inferred entity type is mapped to the corresponding Python `class`. For each predicate p, a member attribute is added to its corresponding domain class, where the values are instances of the range class. To accommodate multi-valued relations, these attributes are represented as lists.

Through this mapping, a large volume of raw triples is compressed into a compact set of typed class definitions. This representation preserves the relational structure of the knowledge graph while providing a programmatic interface for the subsequent code generation stage. Consequently, it significantly reduces the token footprint while maintaining high reasoning flexibility for complex queries.

Code Generation. Based on the generated Python classes $\mathcal{C}_t$, this step synthesizes the executable logic required to resolve the subtask. Specifically, the LLM is prompted with the original question Q, the current subtask specification T_t (comprising topic entities, predicates, and parent step references), and the class definitions $\mathcal{C}_t$. The LLM generates Python code that resolves the subtask primarily through entity traversal, constraint filtering, and data collection. The intermediate results into a predefined container (e.g., a results dictionary). Notably, the code generation process leverages the parent step reference in T_t to access outputs from previous iterations, thereby facilitating non-linear reasoning.

By operating over the abstract class interface rather than raw triples, the generated code remains concise and less prone to the hallucinations typically associated with long-context factual injection. The final execution of this code yields the subtask result R_t, which is then appended to the execution trace $\mathcal{E}_t$.

4.3 Executing

To get the execution result, the module provides a environment to instantiate the Python classes on $\mathcal{G}_{sub}$ and run the codes on it. To ensure robustness, the module incorporates a self-correction loop that allows for iterative refinement of the generated logic based on runtime feedback.

Instantiation and Execution. The execution is conducted within a restricted Python sandbox. For each subtask T_t, the executor concatenates primarily the class definitions $\mathcal{C}_t$, instantiation code, and the operation code. During execution, entities are instantiated into Python objects, and their attributes are populated according to the retrieved triples in $\mathcal{G}_{sub}$. This process produces an in-memory object set $\mathcal{O}_t$ that supports direct attribute traversal, effectively aligning graph-based reasoning with object-oriented execution.

The executor returns either a valid result R_t or an error feedback signal *fb* (e.g., an exception traceback). Notably, CoG treats an empty result as a failure case, as it typically indicates a logical mismatch between the generated code and the underlying KG constraints.

Self-correction and Early Termination. Upon receiving an execution failure or an empty output, the feedback *fb* is fed back to the LLM to trigger code regeneration. This self-correction loop continues for up to N retries. If a valid result R_t is still unobtainable after N attempts, the system performs an early termination and returns the most recent successful candidate answer, thereby preventing unproductive reasoning cycles. Otherwise, a successful pair (T_t, R_t) is appended to the execution trace $\mathcal{E}_t$.

Method	WebQSP	CWQ	GrailQA			
			Overall	I.I.D.	Compositional	Zero-shot
Fine-tuning Methods						
UniKGQA	77.2	51.2	-	-	-	-
TIARA	75.2	-	73.0	87.8	69.2	68.0
KG-Agent	83.3	72.2	-	-	-	-
RoG	85.7	62.6	-	-	-	-
DeCAF	82.1	70.4	-	-	-	-
Pangu	-	-	75.4	84.4	74.6	71.6
FlexKBQA	-	-	62.8	71.3	59.1	60.6
Prompting Methods						
IO Prompting + Qwen3-Coder-30B-A3B	63.1	42.6	33.4	33.5	23.0	37.1
IO Prompting + DeepSeekV3.2	75.3	52.9	35.7	34.8	26.0	39.6
Readi + GPT-4.1-mini	80.9	60.2	71.7	67.5	60.1	77.6
ReKnoS + GPT-4o-mini	83.8	66.8	80.5	-	-	-
SRP + GPT-4.1-mini	83.6	69.0	78.8	75.8	62.6	85.8
PoG + Qwen3-Coder-30B-A3B	82.9	64.1	76.8	77.9	61.1	81.9
PoG + DeepSeekV3.2	83.9	72.6	71.6	68.4	56.4	78.3
CoG + Qwen3-Coder-30B-A3B	76.0	60.8	82.0	82.5	77.9	83.2
CoG + GPT-4.1-mini	85.4	72.2	89.4	89.0	83.6	91.6
CoG + DeepSeekV3.2	**88.7**	**79.1**	**91.0**	**90.5**	**84.2**	**93.5**

Table 1: Main experimental results on WebQSP, CWQ, and GrailQA. The GrailQA results are broken down into Overall, I.I.D., Compositional, and Zero-shot splits.

5 Experiments

5.1 Experimental Setups

Datasets and Evaluation Metric. To evaluate the effectiveness of PoG on complex KG reasoning, we conduct experiments on three widely used multi-hop KGQA datasets: WebQSP (Yih et al., 2016), CWQ (Talmor and Berant, 2018), and GrailQA (Gu et al., 2021), all based on Freebase (Bollacker et al., 2008). For GrailQA, we select the same test subset as used in ToG and PoG to ensure consistency across experiments. In line with prior work (Sun et al., 2024; Chen et al., 2024; Xu et al., 2024), we report exact match accuracy (Hits@1) as our primary metric. Further details on the datasets can be found in Table 4.

Implementation Details. Since the baseline methods employ different backbone models, we select Qwen3-Coder-30B-A3B (Yang et al., 2025), DeepSeekV3.2 (DeepSeek-AI et al., 2025), and GPT-4.1-mini (OpenAI, 2025) for the convenience of comparison. For subtask decomposition, we use a temperature of 1.2 to encourage exploration, while the evaluation module employs a temperature of 0 to ensure strict consistency. Code generation and correction are performed with a temperature of 0.3 to yield stable and reliable outputs. The exploration depth is set to 2 to cover complete CVT relations, and the breadth is set to 8 (i.e., up to 8 incoming/outgoing edges per hop).

Baselines. We compare CoG against two main categories of methods: fine-tuning methods and prompting methods. For the former, we choose UniKGQA(Jiang et al., 2023b), TIARA(Shu et al., 2022), KG-Agent(Jiang et al., 2024), RoG (Luo et al., 2024b), DeCAF(Yu et al., 2023), Pangu (Gu et al., 2023), and FlexKBQA(Li et al., 2024b) as baselines. For the latter, we choose IO prompting (i.e., the model directly answers questions given some examples)(Cahyawijaya et al., 2024), Readi(Cheng et al., 2024), ReKnoS(Wang et al., 2025), SRP(Zhu et al., 2025), and PoG(Chen et al., 2024) for comparison. Appendix D provides the descriptions of the baselines.

5.2 Main Results

As shown in Table 1, we compare CoG with baseline methods on three widely used datasets. Overall, CoG consistently achieves the best performance across all three datasets, outperforming both fine-tuning and prompting-based approaches. This demonstrates the effectiveness of our approach,

which can be analyzed from three key aspects.

Schema-level coding enhances the manipulation of large-scale factual knowledge. Compared to prompting methods like PoG, ReKnoS, and SRP, which rely on reasoning over raw or pruned factual triples, CoG can process a larger volume of evidence programmatically within a compact context by instantiating facts into class structures. This approach significantly enhances fact manipulation capabilities. On the complex multi-hop CWQ dataset, CoG with DeepSeekV3.2 achieves 79.1% Hits@1, which represents a 6.5% absolute improvement over PoG.

Flexible schema-level coding enhances generalization ability. Unlike frameworks such as KG-Agent or Readi, which are restricted by a fixed set of predefined operators, CoG's schema-level code generation allows the model to generalize to diverse and novel reasoning patterns that static tool-use paradigms cannot accommodate. CoG demonstrates exceptional generalization in the compositional and zero-shot settings of GrailQA. Specifically, CoG with GPT-4.1-mini outperforms Readi by a margin of 23.5% on the compositional subset. On the zero-shot subset, the performance lead over Readi reaches 14.0%.

Backbone capability significantly impacts performance. The design of CoG is strongly correlated with the underlying model's ability to comprehend complex schemas and generate object-oriented code. Owing to the robust workflow design and schema-level coding, CoG achieves competitive results using the Qwen3-Coder-30B-A3B backbone. Meanwhile, the highest performance is reached with powerful backbones like DeepSeekV3.2. This indicates that CoG's ceiling scales with the model's programming proficiency; at the same time, its core mechanism of mapping schemas to classes effectively unlocks structured reasoning potential across a wide range of LLM capabilities.

5.3 Ablation Study

We conduct ablation studies to evaluate the contribution of each major component in CoG. Experiments are performed on three datasets using DeepSeek-V3.2 as the backbone model. Experimental results are shown in Table 2. w/o Iterative Planning denotes using pre-decomposed subtasks instead of dynamic subtasks for each question. w/o Error Correction indicates disabling the code correction mechanism and no longer adjusting based on execution feedback from encountered errors. w/o Code Reasoning represents directly injecting triples/facts into prompts instead of using code generation. Since the number of triples can be extremely large—potentially exceeding the context limits of the LLM—we selected up to 500 triples most relevant to each subtask based on similarity, aiming to match the token volume used in CoG. w/ JSON Abstraction refers to replacing Python class definitions for triple schema structure with JSON-formatted representations.

Method / Variant	WebQSP	CWQ	GrailQA
CoG	**88.7**	**79.1**	**91.0**
w/o Iterative Planning	81.7	75.1	88.9
w/o Error Correction	85.9	67.5	85.9
w/o Code Reasoning	84.9	70.5	81.6
w/ JSON Abstraction	88.4	76.4	87.2

Table 2: Ablation study on three KBQA datasets.

The results demonstrate that all variants lead to a decline in performance, confirming that each major component in CoG is effectively designed. Specifically, the w/o Iterative Planning variant drops up to 7%, suggesting that pre-planned subtasks may not adapt well to intermediate discoveries during actual exploration. Moreover, removing error correction (w/o Error Correction) causes a significant performance decrease on three datasets, indicating that without a correction mechanism, CoG lacks robustness and may fail during exploration. When triples are used for reasoning instead of code generation (w/o Code Reasoning), CoG also experiences a substantial decline in performance, suggesting that LLMs are not suitable for direct reasoning over large-scale factual triples, as context window constraints may prevent coverage of all necessary facts, leading to incomplete reasoning and errors. Finally, replacing Python classes with JSON format (w/ JSON Abstraction) results in only a minor performance drop, suggesting that LLMs have a strong understanding of both Python class structures and JSON formats.

5.4 Analysis of Token Utility Rate

To measure CoG's capability to perform large-scale factual knowledge operations without exceeding the context window, we introduce a key metric, **Token Utility Rate** (**TUR**), which measures the average number of facts that can be operated on per token (including both input and output tokens). Specifically, it is defined as the number of tokens

Dataset	Method	# LLM Call	# Reasoning Facts	Token Utility Rate	Gain
CWQ	PoG	25.1	14.0	8.2×10^{-4}	1.0×
	Ours	**7.0**	**20,276.1**	$\mathbf{9.7 \times 10^{-1}}$	**1189.6**×
WebQSP	PoG	13.0	14.2	1.7×10^{-3}	1.0×
	Ours	**3.8**	**11,213.0**	$\mathbf{9.9 \times 10^{-1}}$	**588.6**×
GrailQA	PoG	7.5	7.3	1.6×10^{-3}	1.0×
	Ours	**4.2**	**8,147.5**	$\mathbf{7.3 \times 10^{-1}}$	**457.4**×

Table 3: Comparison of Token Utility Rate between the proposed CoG methods and baselines.

used divided by the facts involved in reasoning.

As shown in Table 3, CoG can reason over a large amount of factual knowledge without causing context overflow. Especially on CWQ, CoG handles an average of over 20,000 facts per question through code-based operations, achieving a 1189× improvement in TUR. On simpler benchmarks like WebQSP and GrailQA, it processes close to 10,000 facts, which substantially exceeds the scale achievable by PoG. Such large-scale, omission-free, and efficient knowledge processing largely explains CoG's significant performance improvements on this dataset. Additionally, CoG completes the question-answering process with fewer LLM calls. Particularly on WebQSP and GrailQA, it answers simple questions with only around 4 calls. For the more complex CWQ dataset, CoG requires only about 30% of the LLM calls used by PoG. This advantage stems from CoG's ability to flexibly generate task-specific operators for each sub-question, which improves per-sub-question success rates and reduces the number of LLM calls required for complex tasks.

5.5 Case Study

As shown in Figure 3, we specifically illustrate an example to compare the reasoning processes and results of PoG and CoG. The complex question "Who was the Governor of Ohio that held his position before 1-10-2011?" involves two subtasks: "Find the governors of Ohio" and "Filter for those whose term started before 2011-01-10". The KG contains extensive information about Ohio and its officeholders. When faced with such a large number of facts, PoG reduces the reasoning context by extensively pruning, which ultimately leads to the loss of critical information and results in errors.In contrast, CoG abstracts entities into a small number of class structures and performs reasoning via code generation and execution. This design preserves key information and substantially reduces the reasoning context, ultimately leading to the correct answer.

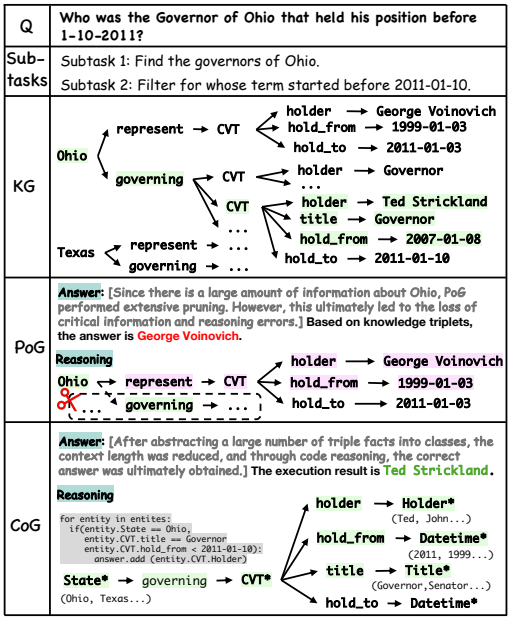

Figure 3: An example for comparing PoG and CoG in answering complex questions.

Additionally, Appendix F demonstrates CoG's flexibility in creating novel operators and its robustness in performing code error correction.

6 Conclusion

In this paper, we introduced a novel LLM-KG framework named Code-on-Graph (CoG), which enhances the reasoning capability of LLMs by incorporating KGs. Specifically, we designed CoG to address complex problems through an iterative process of Planning, Coding, and Executing. We addressed the inflexibility and unscalability of existing LLM-KG methods through these steps.Extensive experiments demonstrated the effectiveness and efficiency of CoG, which achieved state-of-the-art performance without any fine-tuning.

Limitations

CoG is able to better leverage the programming capabilities of large language models to construct diverse operators for solving complex question-answering tasks. However, its performance is still constrained by the coding strength of the underlying model. When instantiated with a comparatively less powerful coding model such as Qwen3-Coder-30B-A3B, the advantages of CoG are not fully manifested. In addition, our experiments only cover a part of the LLMs. More state-of-the-art LLMs remain to be further studied.

References

Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary G. Ives. 2007. Dbpedia: A nucleus for a web of open data. In *The Semantic Web, 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007*, volume 4825 of *Lecture Notes in Computer Science*, pages 722–735. Springer.

Kurt D. Bollacker, Colin Evans, Praveen K. Paritosh, Tim Sturge, and Jamie Taylor. 2008. Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*, pages 1247–1250. ACM.

Samuel Cahyawijaya, Holy Lovenia, and Pascale Fung. 2024. LLMs are few-shot in-context low-resource language learners. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 405–433, Mexico City, Mexico. Association for Computational Linguistics.

Liyi Chen, Panrong Tong, Zhongming Jin, Ying Sun, Jieping Ye, and Hui Xiong. 2024. Plan-on-graph: Self-correcting adaptive planning of large language model on knowledge graphs. *Preprint*, arXiv:2410.23875.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2023. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Preprint*, arXiv:2211.12588.

Sitao Cheng, Ziyuan Zhuang, Yong Xu, Fangkai Yang, Chaoyun Zhang, Xiaoting Qin, Xiang Huang, Ling Chen, Qingwei Lin, Dongmei Zhang, Saravan Rajmohan, and Qi Zhang. 2024. Call me when necessary: Llms can efficiently and faithfully reason over structured environments. *Preprint*, arXiv:2403.08593.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding language models in symbolic languages. *Preprint*, arXiv:2210.02875.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenhao Xu, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, and 245 others. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *Preprint*, arXiv:2512.02556.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models. *Preprint*, arXiv:2211.10435.

Yu Gu, Xiang Deng, and Yu Su. 2023. Don't generate, discriminate: A proposal for grounding language models to real-world environments. *Preprint*, arXiv:2212.09736.

Yu Gu, Sue Kase, Michelle Vanni, Brian Sadler, Percy Liang, Xifeng Yan, and Yu Su. 2021. Beyond i.i.d.: Three levels of generalization for question answering on knowledge bases. In *Proceedings of the Web Conference 2021*, page 3477–3488. ACM.

Xiaoxin He, Yijun Tian, Yifei Sun, Nitesh V. Chawla, Thomas Laurent, Yann LeCun, Xavier Bresson, and Bryan Hooi. 2024. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering. *Preprint*, arXiv:2402.07630.

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

Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, Yang Song, Chen Zhu, Hengshu Zhu, and Ji-Rong Wen. 2024. Kg-agent: An efficient autonomous agent framework for complex reasoning over knowledge graph. *Preprint*, arXiv:2402.11163.

Jinhao Jiang, Kun Zhou, Wayne Xin Zhao, and Ji-Rong Wen. 2023b. Unikgqa: Unified retrieval and reasoning for solving multi-hop question answering over knowledge graph. *Preprint*, arXiv:2212.00959.

Tianle Li, Xueguang Ma, Alex Zhuang, Yu Gu, Yu Su, and Wenhu Chen. 2023a. Few-shot in-context learning for knowledge base question answering. *Preprint*, arXiv:2305.01750.

Xiaoxi Li, Yujia Zhou, and Zhicheng Dou. 2023b. Unigen: A unified generative framework for retrieval and question answering with large language models. *Preprint*, arXiv:2312.11036.

Yading Li, Dandan Song, Changzhi Zhou, Yuhang Tian, Hao Wang, Ziyi Yang, and Shuhao Zhang. 2024a. A framework of knowledge graph-enhanced large language model based on question decomposition and atomic retrieval. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 11472–11485, Miami, Florida, USA. Association for Computational Linguistics.

Zhenyu Li, Sunqi Fan, Yu Gu, Xiuxing Li, Zhichao Duan, Bowen Dong, Ning Liu, and Jianyong Wang. 2024b. Flexkbqa: A flexible llm-powered framework for few-shot knowledge base question answering. *Preprint*, arXiv:2308.12060.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *Preprint*, arXiv:2304.09842.

Haoran Luo, Haihong E, Zichen Tang, Shiyao Peng, Yikai Guo, Wentai Zhang, Chenghao Ma, Guanting Dong, Meina Song, Wei Lin, Yifan Zhu, and Anh Tuan Luu. 2024a. ChatKBQA: A generate-then-retrieve framework for knowledge base question answering with fine-tuned large language models. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 2039–2056, Bangkok, Thailand. Association for Computational Linguistics.

Linhao Luo, Yuan-Fang Li, Gholamreza Haffari, and Shirui Pan. 2024b. Reasoning on graphs: Faithful and interpretable large language model reasoning. *Preprint*, arXiv:2310.01061.

OpenAI. 2025. Openai.

Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *Preprint*, arXiv:1910.01108.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Preprint*, arXiv:2302.04761.

Jie Shi, Bo Xu, Jiaqing Liang, Yanghua Xiao, Jia Chen, Chenhao Xie, Peng Wang, and Wei Wang. 2025. Gen-SQL: Efficient text-to-SQL by bridging natural language question and database schema with pseudo-schema. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 3794–3807, Abu Dhabi, UAE. Association for Computational Linguistics.

Yiheng Shu, Zhiwei Yu, Yuhan Li, Börje F. Karlsson, Tingting Ma, Yuzhong Qu, and Chin-Yew Lin. 2022. Tiara: Multi-grained retrieval for robust question answering over large knowledge bases. *Preprint*, arXiv:2210.12925.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel M. Ni, Heung-Yeung Shum, and Jian Guo. 2024. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. *Preprint*, arXiv:2307.07697.

Dídac Surís, Sachit Menon, and Carl Vondrick. 2023. Vipergpt: Visual inference via python execution for reasoning. *Preprint*, arXiv:2303.08128.

Alon Talmor and Jonathan Berant. 2018. The web as a knowledge-base for answering complex questions. *Preprint*, arXiv:1803.06643.

Lei Wang, Yi Hu, Jiabang He, Xing Xu, Ning Liu, Hui Liu, and Heng Tao Shen. 2023. T-sciq: Teaching multimodal chain-of-thought reasoning via mixed large language model signals for science question answering. *Preprint*, arXiv:2305.03453.

Song Wang, Junhong Lin, Xiaojie Guo, Julian Shun, Jundong Li, and Yada Zhu. 2025. Reasoning of large language models over knowledge graphs with super-relations. *Preprint*, arXiv:2503.22166.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. Chain-of-thought prompting elicits reasoning in large language models. *Preprint*, arXiv:2201.11903.

Yao Xu, Shizhu He, Jiabei Chen, Zihao Wang, Yangqiu Song, Hanghang Tong, Guang Liu, Kang Liu, and Jun Zhao. 2024. Generate-on-graph: Treat llm as both agent and kg in incomplete knowledge graph question answering. *Preprint*, arXiv:2404.14741.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Wen-tau Yih, Matthew Richardson, Chris Meek, Ming-Wei Chang, and Jina Suh. 2016. The value of semantic parse labeling for knowledge base question answering. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 201–206, Berlin, Germany. Association for Computational Linguistics.

Donghan Yu, Sheng Zhang, Patrick Ng, Henghui Zhu, Alexander Hanbo Li, Jun Wang, Yiqun Hu, William Wang, Zhiguo Wang, and Bing Xiang. 2023. Decaf: Joint decoding of answers and logical forms for question answering over knowledge bases. *Preprint*, arXiv:2210.00063.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang,

Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2025. A survey of large language models. *Preprint*, arXiv:2303.18223.

Jiajun Zhu, Ye Liu, Meikai Bao, Kai Zhang, Yanghai Zhang, and Qi Liu. 2025. Self-reflective planning with knowledge graphs: Enhancing llm reasoning reliability for question answering. *Preprint*, arXiv:2505.19410.

Appendix

A Dataset Details

Dataset	Answer Format	Train	Test	KB
WebQSP	Enity/Number	3,098	1,639	Freebase
CWQ	Entity	27,734	3,531	Freebase
GrailQA	Entity/Number	44,337	1,000	Freebase

Table 4: Dataset Statistics. Due to the large size of the GrailQA test set (6,763 samples) and the associated computational costs, we follow the practice of ToG and PoG by using the same randomly selected subset of 1,000 samples for our evaluation.

In this work, we employ three complex multi-hop KGQA datasets: WebQSP (WebQuestionsSP) (Yih et al., 2016), ComplexWebQuestions (CWQ) (Talmor and Berant, 2018), and GrailQA (Gu et al., 2021). Table 4 summarizes their key statistics.

WebQSP consists of questions from WebQuestions that are answerable via Freebase. The dataset evaluates I.I.D. generalization, containing 34 logical form templates and involving 2,461 entities. It defines 628 relations in total.

ComplexWebQuestions (CWQ) extends WebQSP by introducing four categories of complex questions: conjunction, composition, comparative, and superlative. It is structured over 174 logical form templates, includes 11,422 entities, and covers 845 relations.

GrailQA is a large-scale and diverse KGQA benchmark constructed on Freebase, designed to evaluate model generalization across three distinct subsets based on different settings: I.I.D., compositional, and zero-shot.

B Prompts

Figure 4 presents the prompt related to the planning stage, and Figure 5 presents the prompts related to the coding and executing stages.

C Algorithm

Algorithm 1 illustrates the workflow of CoG. At iteration t, the framework proceeds as follows. For $t > 1$, a termination evaluator $\mathcal{M}_{eval}$ outputs a Boolean flag $c_t \in \{\texttt{True}, \texttt{False}\}$ indicating whether to continue exploration. If the generator determines that no further decomposition is required, it emits the special token `END_OF_TASK`, which terminates the overall process. If $c_t = \texttt{True}$, a subtask generator $\mathcal{M}_{gen}$ produces the next subtask T_t. The system then retrieves a constrained subgraph $\mathcal{G}_{sub}$, abstracts it into Python classes and instantiated objects, and asks an LLM to generate Python code that is executed in a sandbox. If execution fails, the error signal is fed back for iterative code regeneration. The latest successful output is tracked as the current best answer A.

D Baseline Descriptions

In this section, we provide supplementary introductions to baseline methods used for comparison. We categorize them into two groups: Fine-tuning Methods and Prompting Methods.

D.1 Fine-tuning Methods

UniKGQA (Jiang et al., 2023b) integrates retrieval and reasoning into a single model architecture, utilizing a pre-trained language model for semantic matching between questions and relations, and a matching information propagation module to diffuse relevance signals across the graph. To handle different search scales, the approach introduces abstract subgraphs during retrieval to reduce node complexity. The model is trained via shared pre-training on question-relation matching and stage-specific fine-tuning, effectively bridging the gap between retrieval and reasoning while addressing the challenge of scalable multi-hop QA over large KGs.

TIARA (Shu et al., 2022) enhances pre-trained LMs with multi-grained retrieval from knowledge bases. It retrieves relevant entities, exemplary logical forms, and schema items to provide both semantic and structural context for question answering. The approach further employs constrained decoding during generation to ensure syntactic and semantic correctness of logical forms, addressing challenges in grounding and generalization over large KBs.

KG-AGent (Jiang et al., 2024) is an autonomous agent framework that enables smaller LLMs (e.g., LLaMA-7B) to perform complex multi-hop reasoning over KGs. The framework integrates a

Task Decomposition Prompt

```
You are an expert in navigating, interpreting, and performing complex reasoning over structured
knowledge bases.
Given a Question, a Trajectory (the previous reasoning steps), and the Current Answers, your task is
to decide whether further exploration is required.

You must output a JSON object in the following format:
{
    "continue": <true or false>, // boolean value, whether more exploration is needed
    "reason": "<brief explanation>" // why to continue or stop
}

Requirements:
- 'reason' must be concise, and must justify the decision by jointly considering:
1. the difficulty of the question (e.g., presence of constraints, multi-hop requirements)
2. the soundness of the current responses (whether they match the question intent)
3. the sufficiency of the exploration so far (whether key constraints were checked)

- The explanation should reflect whether:
1. the current responses satisfy the required constraints
2. the trajectory includes both the necessary reasoning steps
3. additional exploration is needed to resolve missing or unexamined conditions

Below are examples for reference:
<Examples>

Your task:
Question: <Question>
Trajectory: <Trajectory>
Current responses: <Current Responses>
```

Task Evaluation Prompt

```
You are an expert in navigating, interpreting, and performing complex reasoning over structured
knowledge bases.
Given a question, its task history, and results, you must generate the next task by thinking step by
step and outputting in JSON format.

JSON format:
- thought: your thinking process
- step: int, refers to the subtask index
- task: next task
- predicates: list, potentially semantically related predicates in the knowledge base
- topic_mids: list, max length of 2, the mids of topic entities from provided to complete the
subtask
- parent_step: int list, the origin of current task from previous step, empty array if no parent
task

Please follow the rules:
1. For ambiguous questions, you may rewrite the query by connecting alternatives with "or" to cover
as many potential queries as possible.
2. Before generating the subtask, you must think briefly about the question.
3. topic_mids should ONLY be selected from the provided topic entities.


Examples for task decomposition sequence:
<Examples>

Question: <Question>
Provided topic entities: <Topic Entities>
Previous Steps: <Previous Steps>
```

Figure 4: Task decomposition and evaluation prompts.

Code Generation Prompt

```
You are good at coding.
Given a task and relevant Python classes, you must generate simple and efficient Python code between
'# Your code starts here:' and '# Your code ends here:' to complete the task.

Rules:
1. CVT's '_mid' is NOT ALLOWED to be collected as a result since CVT refers to an intermediate entity.
2. If a short explanation is needed, it should be in a code comment.
3. All attributes are of String type, such as the '_value' of the 'Date' class, e.g., '"2002-06-05"'.
4. If you think the task is ambiguous, you can make multiple attempts and ensure you explore
potential correct results.
5. You must traverse all entities completely (use 'break' statements with caution) to avoid missing
potential answers.
6. When traversing all entities and using '_mid' for judgment, you must first check for the presence
of the attribute using 'hasattr' or 'isinstance', as not all classes have the '_mid' attribute.
7. Reading the comments on class attributes before coding is highly recommended.

Result Format:
1. All results must be collected in 'result_dict'. Additionally, always collect the entity's '_mid'
and '_name' in 'result_dict['detailed_results']'. '_mid' can be directly obtained from the predefined
'mid_name_map'.
2. If the query concerns a literal, date, number, etc., you should also collect them in
'result_dict["direct_results"]'.

Examples:
<Examples>

Your task:
Question: <Question>
Task: <Task>
Topic entities: <Topic entities>
Code: <Classes Defeination Code>
```

Code Correction Prompt

```
You are good at logic analysis and coding. You aim to fix a specific program to complete the task.

Rules:
1. CVT's '_mid' is NOT ALLOWED to be collected as a result since CVT refers to an intermediate
entity.
2. If a short explanation is needed, it should be in a code comment.
3. All attributes are of String type, such as the '_value' of the 'Date' class, e.g., '"2002-06-05"'.
4. If you think the task is ambiguous, you can make multiple attempts and ensure you explore
potential correct results.
5. You must traverse all entities completely (use 'break' statements with caution) to avoid missing
potential answers.
6. When traversing all entities and using '_mid' for judgment, you must first check for the presence
of the attribute using 'hasattr' or 'isinstance', as not all classes have the '_mid' attribute.
7. Reading the comments on class attributes before coding is highly recommended.

Result Format:
1. All results must be collected in 'result_dict'. Additionally, always collect the entity's '_mid'
and '_name' in 'result_dict['detailed_results']'. '_mid' can be directly obtained from the
predefined 'mid_name_map'.
2. If the query concerns a literal, date, number, etc., you should also collect them in
'result_dict["direct_results"]'.

Potential errors for reference:
1. Forging entity's mid: Do not use the mid of a topic entity that has not been provided.
2. Logical error: Simply revise the logic.
3. Taking the _value (instead of _mid) of the entity as result: You should use only the '_mid' as
result.
4. Repeatedly searching for the answer from the previous step: The relevant constraints from the
previous step have already been completed, and in the current step, these repetitive steps must be
skipped to explore or apply new constraints.
5. Conditional judgment exits prematurely: Member variables receive lists of entities; as long as
one entity meets the condition, it is considered satisfied. Premature breaks should be avoided.
6. Overly precise _value matching: String equality matching is a very strict; the matching criteria
can be appropriately relaxed, and multiple fuzzy matching patterns can be added for date, number,
and code matching, etc.
7. Reverse relation: Use the wrong the direction of the relation. Such as (contained by, contain),
(sub, parent), ect.
8. Detailed results key error: The key in retailed_results should be a certain mid.

Examples:
<Examples>

Your task:
Question: <Question>
Task: <Task>
Topic entities: <Topic entities>
Code to be fixed: <Code>
```

Figure 5: Code generation and correction prompts.

Algorithm 1 CoG

Require: Question Q, knowledge graph $\mathcal{G}$, max iterations $T_{\max}$, max debug retries N
Ensure: Final answer A

$\mathcal{T}_0 \leftarrow [\,]$; $\mathcal{E}_0 \leftarrow [\,]$; $A \leftarrow \emptyset$
for $t = 1$ **to** $T_{\max}$ **do**
  if $t > 1$ **then**
    $c_t \leftarrow \mathcal{M}_{eval}(Q, \mathcal{T}_{t-1}, R_{t-1})$ {c_t: continue?}
    if $c_t =$ `False` **then**
      break
    end if
  end if
  $T_t \leftarrow \mathcal{M}_{gen}(Q, \mathcal{E}_{t-1})$
  if $T_t =$ `END_OF_TASK` **then**
    break
  end if
  $\mathcal{G}_{sub} \leftarrow \textsc{Retrieve}(\mathcal{G}, T_t)$
  $(\mathcal{C}_t, \mathcal{O}_t) \leftarrow \textsc{Abstract}(\mathcal{G}_{sub})$ {$\mathcal{C}_t$: class definitions; $\mathcal{O}_t$: instantiated objects}
  $fb \leftarrow$ `None`; $R_t \leftarrow \emptyset$; $ok \leftarrow$ `False`
  for $i = 1$ **to** N **do**
    $P_{t,i} \leftarrow \textsc{CodeGen}(Q, T_t, \mathcal{C}_t, fb)$
    $(R_t, fb) \leftarrow \textsc{Execute}(P_{t,i}, \mathcal{O}_t)$
    if $fb =$ `None` **and** $R_t \neq \emptyset$ **then**
      $ok \leftarrow$ `True`
      break
    end if
  end for
  if $ok =$ `False` **then**
    break {stop and return the last successful A}
  end if
  $\mathcal{T}_t \leftarrow \mathcal{T}_{t-1} \,\|\, [T_t]$ {append}
  $\mathcal{E}_t \leftarrow \mathcal{E}_{t-1} \,\|\, [(T_t, R_t)]$
  $A \leftarrow R_t$
end for
return A

tuned LLM as a planner, a multifunctional toolbox for KG operations, a KG-based executor, and a knowledge memory module. It employs code-based instruction tuning synthesized from existing KGQA datasets to teach the LLM to autonomously select tools and iteratively reason over KG structures. This approach addresses the limitations of pre-defined LLM-KG interaction strategies and reduces dependency on large, closed-source LLMs.

RoG (Luo et al., 2024b) synergizes LLMs with KGs to address the issues of hallucination and lack of up-to-date knowledge in LLM reasoning. It introduces a planning-retrieval-reasoning framework where RoG first generates relation paths grounded by KGs as faithful plans, then retrieves corresponding reasoning paths from KGs, and finally conducts interpretable reasoning based on these retrieved paths. By distilling structural knowledge from KGs into the LLM through instruction tuning, RoG enables faithful and interpretable multi-hop reasoning while maintaining the flexibility to integrate with arbitrary LLMs during inference.

DecAF (Yu et al., 2023) jointly decodes both logical forms and direct answers within a unified retrieval-augmented sequence-to-sequence model. It linearizes the knowledge base into text passages and employs either sparse (BM25) or dense (DPR) retrieval to fetch relevant context, instead of relying on dedicated entity linking. Using a shared T5-based reader with task-specific prefixes, the model generates logical forms and answers in parallel. The final answer is determined by prioritizing executable logical-form answers when available, otherwise falling back to the directly generated answer. This approach effectively combines the precision of semantic parsing with the robustness of direct generation, while simplifying adaptation across datasets by eliminating the need for specialized entity linking components.

Pangu (Gu et al., 2023) leverages LMs primarily as discriminators rather than generators. In this framework, a symbolic agent interacts with the environment to incrementally enumerate and search for valid, executable candidate plans. The neural LM is employed to score and rank these candidate plans based on their plausibility with respect to the input natural language utterance, guiding the search process. This approach shifts the burden of ensuring grammatical correctness and environmental faithfulness from the LM to the agent, ef-

fectively addressing key challenges in grounding language to complex, real-world environments.

FlexKBQA (Li et al., 2024b) first samples diverse, executable programs (e.g., SPARQL queries) from the knowledge base and uses LLMs to convert them into natural language questions, generating a synthetic dataset for training a lightweight model. To bridge the distribution gap between synthetic data and real user questions, it introduces an execution-guided self-training method that iteratively annotates unlabeled user queries. Additionally, the framework incorporates the inherent reasoning capability of LLMs to further enhance robustness and coverage, thereby reducing reliance on extensive manual annotations while maintaining performance across different domains and query languages.

D.2 Prompting Methods

IO Prompting refers to the approach of providing the LLM with several question-answer exemplars and directly prompting it to generate answers. This method relies solely on the internal knowledge of the large model without introducing additional knowledge for assistance. It is prone to generating hallucinations and suffering from outdated knowledge, highlighting the limitations of LLM-based question answering.

Readi (Cheng et al., 2024) is framework for efficient and faithful reasoning of LLMs over structured environments like knowledge graphs and tables. It leverages the intrinsic planning ability of LLMs to generate an initial reasoning path, which is then directly instantiated on the environment. By invoking LLMs for path editing only when instantiation fails, Readi reduces iterative LLM calls while using instantiation feedback—such as error locations and candidate relations—to guide corrections. This approach enables LLMs to reason over large-scale structured data with fewer interactions while maintaining grounding fidelity.

ReKnoS (Wang et al., 2025) is a novel reasoning framework that enhances the ability of LLMs to perform multi-hop reasoning over knowledge graphs by leveraging the concept of super-relations. Super-relations are groups of semantically related fine-grained relations, which are used to expand the search space and mitigate retrieval failures such as misdirection and depth limitations. The framework iteratively selects and scores candidate super-relations using LLM prompting, enabling simultaneous forward and backward reasoning without exponentially increasing computational cost. By summarizing multiple relational paths into super-relations, ReKnoS effectively addresses the challenge of low retrieval rates in complex knowledge graph question answering tasks.

SRP (Zhu et al., 2025) integrates iterative planning and reflection by first searching for relevant references to guide the process, then generating and checking initial reasoning paths. It performs knowledge retrieval from KGs and employs a self-reflection mechanism to judge and edit the reasoning paths iteratively until a reliable answer is retrieved. The approach addresses the problem of LLM hallucinations and incomplete reasoning in KGQA by combining structured knowledge retrieval with systematic, feedback-driven path refinement.

PoG (Chen et al., 2024) is a novel self-correcting adaptive planning paradigm for LLMs augmented with KGs. PoG addresses the limitations of existing KG-augmented LLM methods, such as fixed exploration breadth, irreversible reasoning paths, and susceptibility to forgetting query conditions. The framework iteratively performs task decomposition into sub-objectives, conducts adaptive path exploration on the KG, maintains a dynamic memory of retrieved subgraphs and reasoning states, and employs a reflection mechanism to evaluate and self-correct erroneous paths. By integrating guidance through sub-objectives, continuous memory updates, and reflective backtracking, PoG enables more flexible, reliable, and efficient graph reasoning, effectively enhancing the planning and correction capabilities of LLMs in complex KG question-answering scenarios.

E Analysis of Code Correction

Analysis of Different Code Correction Attempts. During the generation of reasoning code by the LLM, various errors may occur—such as incorrect variable selection, wrong relation extraction, or mistakenly choosing CVT nodes as answers. These issues are corrected through feedback-driven retries. Therefore, we allow CoG a limited number of attempts. We investigate the relationship between the maximum number of attempts per question and Hits@1.

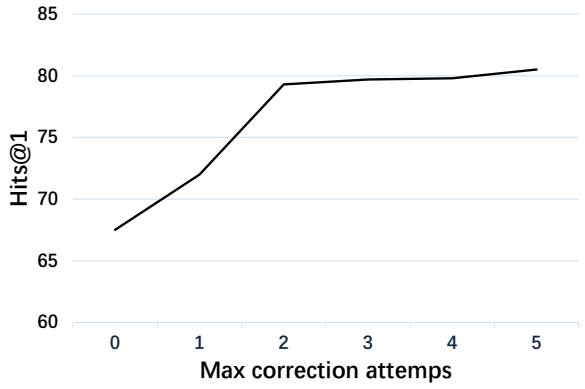

Figure 6: Different Max Correction Attempts and Performance on CWQ

As shown in Figure 6, for the more complex CWQ dataset (where each question requires multi-step reasoning), we vary the maximum attempts from 0 to 5. When set to 0, no error feedback or retry is allowed. The results show that increasing the maximum number of attempts consistently improves CoG's performance. However, considering the additional time and token cost incurred by retries, we recommend setting the maximum attempt limit to 3.

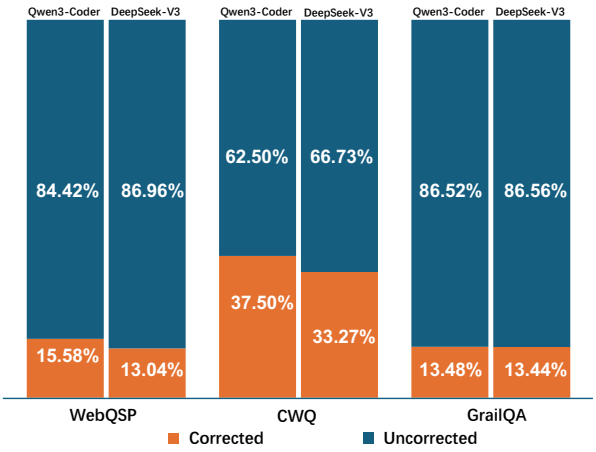

Figure 7: The proportion of cases whether involve code correction with Qwen3-Coder-30B-A3B (Qwen3-Coder in Figure) and DeepSeek-V3.2 (DeepSeek-V3 in Figure) on three datasets. Corrected indicates invocation, while Uncorrected indicates the opposite.

The analysis of code correction invocation. Figure 7 shows the proportion of code correction invocations by CoG using different models across three datasets. Corrected indicates invocation, while Uncorrected indicates the opposite. Overall, a significant number of samples invoked the error correction mechanism. CoG's proportion of error correction invocations on WebQSP and CWQ is lower than that on CWQ, approximately only half of the latter. This is because CWQ has more complex question patterns and greater reasoning difficulty, making it more prone to errors. Examining the performance of different models within each dataset individually reveals that DeepSeek-V3.2, which possesses stronger reasoning and coding capabilities, has a lower proportion of error correction invocations compared to the relatively weaker Qwen3-Coder. This indicates that more powerful models exhibit better stability.

Figure 8 shows the performance of the samples after the code correction mechanism is applied, specifically whether the final answer is hit or not. It can be observed that CoG, using both Qwen3-Coder-30B-A3B and DeepSeekV3.2, successfully corrected many samples, especially the latter, which corrected more than half of the samples. This demonstrates the powerful robustness of CoG.

F Case Study

As illustrated in figure 3, When performing code reasoning, CoG demonstrates rich logical compositions. In Figure 9, we typically select three representative scenarios in which CoG generates logical operations, namely Argmax, Argmin, and Union.

For the question "Which school with the latest founding date did James Baldwin attend?", CoG compares the founding dates of the schools and selects the one with the maximum date. For the question "Holding his governmental position from earliest, who founded New York University?", CoG identifies the founder with the earliest starting date by comparing the dates of holding governmental positions. For the question "What major trading partner of China is the home of Monteith's Lager beer?", it is necessary to identify countries that either export to China or import from China. CoG accomplishes this operation using OR logic.

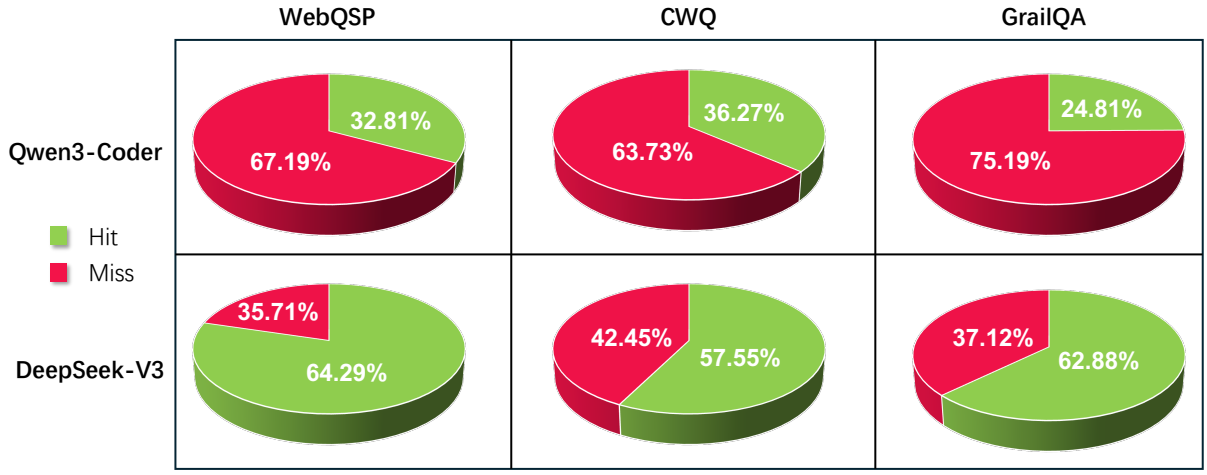

Figure 8: The hit rate of the corrected cases for Qwen3-Coder-30B-A3B (Qwen3-Coder in Figure) and DeepSeek-V3.2 (DeepSeek-V3 in Figure).

Argmax

Question: Which school with the latest founding date did James Baldwin attend?

Find the schools attended by James Baldwin. → Find the founding dates of each school from step 1. → Identify which school has the latest founding date from step 2

Class Definations

```
class EducationalInstitution:
  # entities like The New School(m.06thjt) , etc.
  def __init__(self, _mid: str = None,
    education_educational_institution_mascot: List['SchoolMascot'] = None,
    ...
    organization_organization_date_founded: List['Datetime'] = None):
  self._mid = _mid
  ...
  # describes a school's mascot
  self.education_educational_institution_mascot =
education_educational_institution_mascot or []

  # describes the date this organization first came into being
  self.organization_organization_date_founded =
organization_organization_date_founded or []

class Datetime:
...
```

Reasoning Code

```
...
latest_date = None
latest_schools = [ ]
for mid, date_str in founding_dates.items():
  if not date_str:
    continue
  # discover a later date and reset the list
  if latest_date is None or date_str > latest_date:
    latest_date = date_str latest_schools = [mid]
  # same latest date, add to the list.
  elif date_str == latest_date:
    latest_schools.append(mid)
...
```

Argmin

Question: Holding his governmental position from earliest, who founde New York University?

Find the founders of New York University. → Find the governmental positions and their start dates from step 1. → Compare the earliest governmental position start dates from step 2

Class Definations

```
class GovernmentPositionHeldCVT:
  # CVT nodes, which refer to an intermediate step
  def __init__(self, _mid: str = None,
  ...
    government_government_position_held_from: List['Datetime'] = None,
    government_government_position_held_to: List['Datetime'] = None):
  # describes the date when the officeholder started working in this
government position
  self._mid = _mid

  # describes the date when the officeholder ceasedworking in this
government position
  self.government_government_position_held_from =
government_government_position_held_from or []

  # describes the date when the officeholder ceasedworking in this
government position
  self.government_government_position_held_to =
government_government_position_held_to or []

class Datetime:
...
```

Reasoning Code

```
...
if "governmental_positions" in founder_data:
  earliest_date = None
  position_details = []
  for position in
founder_data["governmental_positions"]:
    if "start_dates" in position:
      for start_date in position["start_dates"]:
        # Compare dates as strings (YYYY-MM-DD format
allows string comparison)
        if start_date:
          if earliest_date is None or start_date <
earliest_date:
            earliest_date = start_date
...
```

Union

Question: What major trading partner of China is the home of Monteith's Lager beer?

Find the country or location of origin for Monteith's Lager beer. → Find the major trading partners of China. → Find the country that is both the origin of Monteith's Lager and a major trading partner of China.

Class Definations

```
class ImportsAndExportsCVT:
  # CVT nodes, which refer to an intermediate step
  def __init__(self, _mid: str = None,
    ...
    location_imports_and_exports_exported_to:
List['StatisticalRegion'] = None,
    location_imports_and_exports_imported_from:
List['StatisticalRegion'] = None):
  self._mid = _mid

  # describes the exported to of imports and exports
  self.location_imports_and_exports_exported_to =
location_imports_and_exports_exported_to or []

  # describes the imported from of imports and exports
  self.location_imports_and_exports_imported_from =
location_imports_and_exports_imported_from or []

class StatisticalRegion:
...
```

Reasoning Code

```
...
for region in entity.location_imports_and_exports_exported_to:
  ...
    # Then the imported_from regions are China's trading partners (exporters
to China)
    if hasattr(entity, 'location_imports_and_exports_imported_from'):
      for partner in entity.location_imports_and_exports_imported_from:
        if hasattr(partner, '_mid') and partner._mid != china_mid:
          trading_partners.add(partner._mid)
    # Check if China is imported_from (i.e., China exports to some country)
    ...
        # Then the exported_to regions are China's trading partners
(importers from China)
      if hasattr(entity, 'location_imports_and_exports_exported_to'):
        for partner in entity.location_imports_and_exports_exported_to:
          if hasattr(partner, '_mid') and partner._mid != china_mid:
            trading_partners.add(partner._mid)
...
```

Figure 9: Three representative operators in the CWQ dataset(i.e., Argmax, Argmin, and Union). For each question, we present the scenario corresponding to one of the subtasks. The class definitions of the subtask are shown on the left, while the corresponding operation code is displayed on the right. The parts marked in blue represent the core information relevant to the operation.