CodeSteer: Symbolic-Augmented Language Models via Code/Text Guidance

Yongchao Chen [1 2] **Yilun Hao** [1] **Yueying Liu** [3] **Yang Zhang** [4] **Chuchu Fan** [1]

Abstract

Existing methods fail to effectively steer Large Language Models (LLMs) between textual reasoning and code generation, leaving symbolic computing capabilities underutilized. We introduce CodeSteer, an effective method for guiding LLM code/text generation. We construct a comprehensive benchmark SymBench comprising 37 symbolic tasks with adjustable complexity and also synthesize datasets of 12k multi-turn guidance/generation trajectories and 5.5k guidance comparison pairs. We fine-tune the Llama-3-8B model with a newly designed multi-turn supervised fine-tuning (SFT) and direct preference optimization (DPO). The resulting model, CodeSteerLLM, augmented with the proposed symbolic and self-answer checkers, effectively guides the code/text generation of larger models. Augmenting GPT-4o with CodeSteer raises its average performance score from 53.3 to 86.4, even outperforming the existing best LLM OpenAI o1 (82.7), o1-preview (74.8), and DeepSeek R1 (76.8) across all 37 tasks (28 seen, 9 unseen). Trained for GPT-4o, CodeSteer demonstrates superior generalizability, providing an average 41.8 performance boost on Claude, Mistral, and GPT-3.5. CodeSteer-guided LLMs fully harness symbolic computing to maintain strong performance on highly complex tasks. Models, Datasets, and Codes are available at https://github.com/yongchao98/CodeSteer-v1.0 and https://huggingface.co/yongchao98.

[1]Massachusetts Institute of Technology, Boston, MA, USA [2]Harvard University, Boston, MA, USA [3]University of Illinois Urbana-Champaign, Urbana, IL, USA [4]MIT-IBM Watson AI Lab, Boston, MA, USA. Correspondence to: Yongchao Chen <yongchaochen@fas.harvard.edu>, Chuchu Fan <chuchu@mit.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

1. Introduction

While the reasoning and planning capabilities of LLMs have improved significantly (Wang et al., 2024; Chen et al., 2024c; Li et al., 2023), they still fail in ostensibly simple tasks (Zhou et al., 2024a). Crucially, many tasks in existing benchmarks—such as Blocksworld (Valmeekam et al., 2024) and Game 24 (Zhou et al., 2023b)—can be completely solved with code solutions. Text-based reasoning excels at semantic understanding and commonsense inference but is less suited for exact computation, symbolic manipulation, optimization, and algorithmic processing (Valmeekam et al., 2022). In contrast, symbolic computing via code generation is adept at handling rigorous operations and can easily leverage specialized tools (e.g., equation solvers). In many tasks, prompting LLMs to generate and execute code outperforms purely textual reasoning (Madaan et al., 2022; Liang et al., 2022; Chen et al., 2022).

A key challenge is guiding LLMs to decide when to rely on textual reasoning versus programmatic solutions, given that most input questions lack explicit cues about which approach is best. Recent OpenAI GPT models address this by providing a Code Interpreter module, allowing the model to iteratively generate and execute code, then further reason with the output (Achiam et al., 2023). Multi-agent frameworks like AutoGen (Wu et al., 2023) adopt a specialized system prompt to steer LLM for code generation when needed. However, recently Chen et al. (2025) finds that all these existing methods struggle to effectively steer between textual reasoning and code generation, failing to fully leverage symbolic computing capabilities.

Our work tries to bridge this gap by developing an assistant framework (CodeSteer) to guide the code/text generation of the LLM solving the task (TaskLLM). By fine-tuning a small model (Llama-3-8B (Dubey et al., 2024)) to be the assistant, we enable large models (GPT-4o (Achiam et al., 2023)) to fully leverage symbolic computing via code generation while preserving other capabilities. Recognizing that iterative "executing and exploring" is the most effective way to solve tasks, we build CodeSteer to generate prompts that guide the TaskLLM through multiple turns of interaction before finalizing answers.

To achieve a comprehensive evaluation, we gather and develop a benchmark with 37 symbolic tasks, referred as Sym-

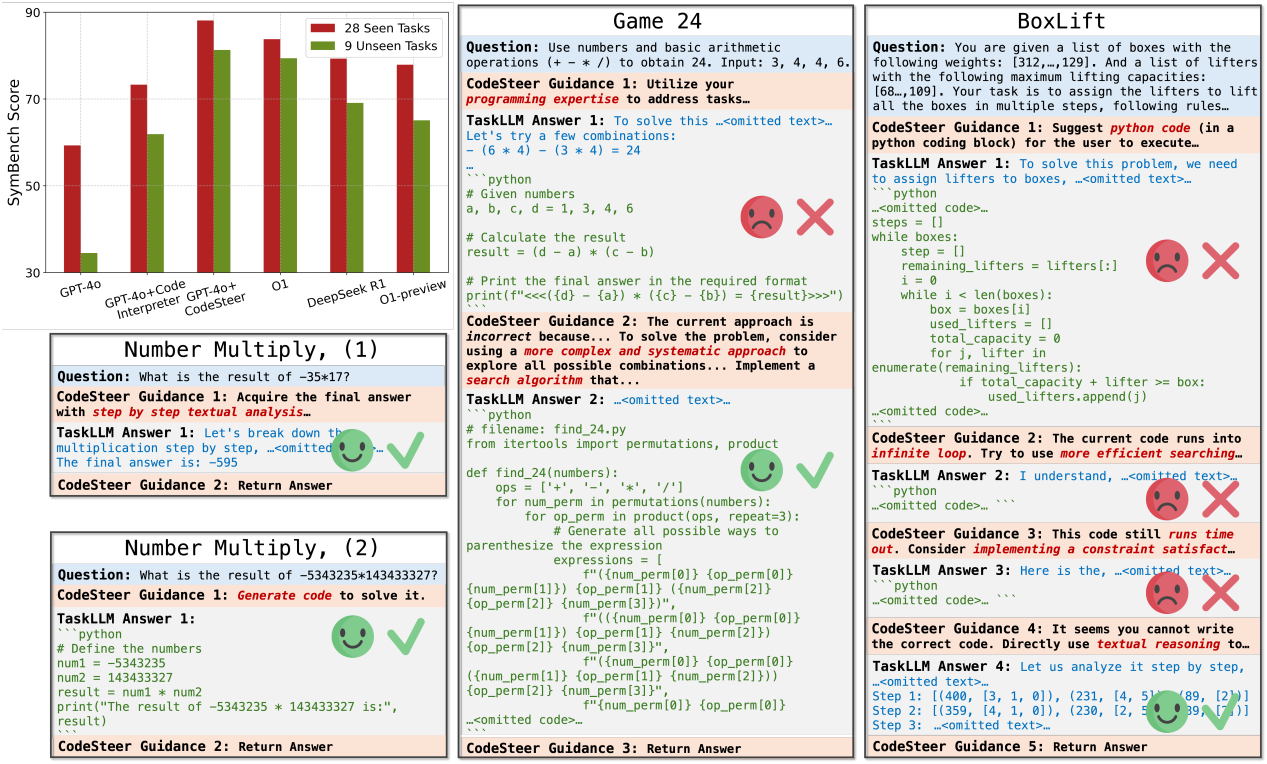

Figure 1: Examples and performance of CodeSteer on guiding LLM code/text generation to integrate symbolic computing. At each interaction with TaskLLM, it reviews current and previous answers, then provides guidance for the next turn. CodeSteer returns final answers when it deems them ready. With CodeSteer, GPT-4o outperforms OpenAI Code Interpreter, o1, and o1-preview models.

Bench. On SymBench, augmenting GPT-4o with CodeSteer greatly improves its average performance score from 53.3 to 86.4, even outperforming the current leading pure-text model, OpenAI o1 (82.7) (Jaech et al., 2024) and DeepSeek R1 (76.8) (Guo et al., 2025). Although trained for GPT-4o, CodeSteer shows great generalizability, delivering an average 41.8 performance gain on Claude-3-5-Sonnet, Mistral-Large, and GPT-3.5. By fully leveraging symbolic computing, CodeSteer-guided LLMs maintain strong performance on highly complex tasks even when o1 fails in all testing cases. Our key contributions are:

1) Developing and publishing SymBench: Prior works by Chen et al. (2025) and Gui et al. (2024) gathered and developed 14 and 31 tasks, respectively, targeting challenges in computation, symbolic manipulation, logic, optimization, spatial reasoning, and constrained planning. However, neither study published the complete code for question/solution synthesis or the full datasets. From these 45 tasks, we select 37 that remain challenging for GPT-4o and redevelop their generation code to produce samples with adjustable complexity. We refer to this newly published benchmark as SymBench.

2) New methods for dataset construction and model fine-tuning of SFT and DPO: We fine-tune Llama-3-8B with the synthesized datasets of 12k multi-turn guidance/generation trajectories (SFT) and 5.5k guidance comparison pairs (DPO). Unlike standard multi-step settings, in CodeSteer's multi-turn guidance, the TaskLLM outputs a complete answer each turn rather than only at the end. Consequently, we introduce novel components to both the dataset construction and training processes for SFT and DPO, such as data synthesis of dynamic guidance adaptation, emphasis on the final two turns in SFT, comparison score design, and efficient answer sampling in DPO. These modifications result in better performance. Both the final CodeSteer model and created datasets will be released.

3) Symbolic checker and self-answer checker: Observing that TaskLLM frequently produces text-like code that hard-codes answers, neglecting efficient symbolic computation, we introduce a Symbolic Checker to help CodeSteerLLM evaluate code complexity and efficiency. Since most reasoning and planning tasks can be better verified with coding, we add a Self-answer Checker for better judgment of answer correctness of CodeSteerLLM. These two new checkers have been proven to significantly improve the efficiency of

dataset synthesis and CodeSteerLLM fine-tuning.

4) Proposed CodeSteer Outperforms Nine Baselines and o1: CodeSteer's superior performance highlights the importance of enhancing LLM reasoning and planning with symbolic computing. This also demonstrates the potential for steering large models to generate smarter code and text by leveraging specialized smaller models.

2. Symbolic Tasks and SymBench

Challenges in Code/Text Choices For tasks requiring computation, symbolic manipulation, logic, optimization, spatial reasoning, and constrained planning, coding-based symbolic computing is often more effective than text-based approaches. However, Chen et al. (2025) found that steering LLM code/text generation poses significant challenges, even in tasks with apparent symbolic characteristics. The main bottlenecks are: 1) Deciding whether code or text is simpler depends on task type, task complexity, and the LLM's capabilities, which is hard to judge (see Appendix Sec. A). 2) LLM-generated code often appears as text-like scripts that merely hard-code answers rather than enabling efficient symbolic computation, echoing the phenomenon described in Yang et al. (2024) (see Appendix Sec. B).

SymBench Chen et al. (2025) and Gui et al. (2024) collected 14 and 31 tasks with symbolic factors from various benchmarks such as Suzgun et al. (2022); Chen et al. (2024d); Yao et al. (2024); Cobbe et al. (2021); Hendrycks et al. (2021), but their question-generation code and complete datasets remain private. We redevelop the generation code to automatically synthesize questions with adjustable complexity. Our resulting set of 37 tasks covers reasoning, planning, and execution, testing competencies in mathematics, spatial reasoning, logic, order reasoning, optimization, and search. Details and categorization are provided in Appendix Sec. C and Table 5.

3. CodeSteer Framework

Fig 1 illustrates how CodeSteer guides the LLM's code/text generation. At each turn, CodeSteer reviews the TaskLLM's current answer and the guidance/answer history, then decides whether to offer new guidance or finalize the response. It performs three key functions:

1) Initial Method Selection In the first turn, it chooses whether to solve the task with code or text (e.g., use textual reasoning for small-number multiplication, and code for large-number multiplication in the task Number Multiply).

2) Dynamic Adaptation In subsequent turns, it refines guidance or switches methods if issues arise (e.g., encouraging more sophisticated symbolic approaches in Game 24, or switching to textual reasoning after multiple incorrect code attempts in BoxLift).

3) Answer Finalization When Ready

The main components of CodeSteer are as follows:

CodeSteerLLM is the primary model fine-tuned and used to guide TaskLLM in code/text generation. The input prompt formats for the first and subsequent turns are presented in Appendix Sec. D. To facilitate answer evaluation, CodeSteerLLM is equipped with two checkers—Self-answer and Symbolic—whose design is inspired by the inherent features of symbolic tasks.

Self-answer Checker re-queries TaskLLM to generate and execute code for verifying its current answer, then returns the evaluation results and explanations to CodeSteerLLM. Since many symbolic tasks benefit from code-based verification, this approach often provides a more reliable perspective. The prompt format for the Self-answer Checker is provided in Appendix Sec. E.

Symbolic Checker is a rule-based script to analyze the generated code for iteration, search, numeric handling, permutations, and combinations, then returns a complexity summary and score. This helps CodeSteerLLM determine whether the code is sufficiently sophisticated for the task at hand. Since TaskLLM often produces text-like code prone to errors, the Symbolic Checker's complexity assessment aids, but does not solely dictate, CodeSteerLLM's decisions. Further details on the checking code and prompt are in Appendix Sec. F.

Beyond enhancing CodeSteerLLM's performance, the Self-answer and Symbolic Checkers also streamline dataset synthesis for SFT and DPO fine-tuning, as discussed in the following sections.

4. Fine-tuning the CodeSteerLLM

Among the three modules of CodeSteer, the CodeSteerLLM needs to be fine-tuned to perform the complicated task of steering. The fine-tuning is performed on a subset of SymBench. Specifically, we randomly select 28 of the 37 SymBench tasks, using a distinct set of samples without overlap with the test samples. This setup allows us to evaluate CodeSteer on 28 seen tasks (with different test samples) and on the remaining 9 unseen tasks. The fine-tuning consists of two steps. We first fine-tune the Llama-3.1-8B model with SFT, then further optimize it using DPO. Both processes are fine-tuned with full parameter on 4*H100 GPUs for 4-10 epochs. The detailed parameter and hardware settings for fine-tuning and inference processes are discussed in Appendix Sec. H. We synthesize 12k multi-turn guidance/generation trajectories for SFT and 5.5k guidance comparison pairs for DPO. The specific data number for each task is in Appendix Sec. G.

4.1. Multi-turn SFT

To generate supervision data for SFT, we prompt the GPT-4o to serve as both the guiding LLM (i.e., the CodeSteerLLM) and the TaskLLM to generate multiple guidance/generate trajectories. We then filter the trajectories keeping only those that produce correct answers. To improve success rates, CodeSteerLLM's prompt is more detailed and includes pre-set knowledge or hints. To increase dataset diversity and enable dynamic adaptation of guided thoughts, this prompt also has different versions. For example, we may let GPT-4o choose all guidance styles, or enforce transitions from code to text or text to code. We set the maximum guidance turns to be 5 and return the final answer once that limit is reached.

Multi-turn Gradient Cancellation Issue In multi-turn trajectories, the SFT process incorporates gradients from each turn. This can lead to gradient cancellation in the early turns. For example, in one task, both [code, return answer] and [text, code, return answer] produce correct results, so if both trajectories are used for fine-tuning, the SFT cannot learn that code is the better first step.

Data Augmentation To mitigate this issue, we leverage the fact that the final two turns of guidance are most influential, as the TaskLLM produces new answers each turn while earlier turns primarily provide background. Consequently, we augment the SFT dataset by doubling the weights of the final two turns.

4.2. Multi-turn DPO

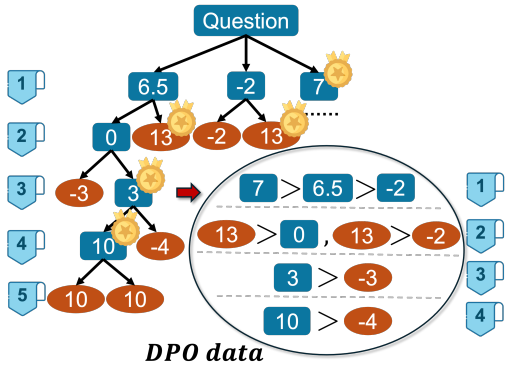

Figure 2: Schematic of multi-turn DPO data sampling: blue squares represent intermediate (non-final) turns, and brown ovals mark finalizing turns. Guidance responses from the same parent node in CodeSteerLLM are compared to generate the DPO data.

Because many correct trajectories in the SFT dataset are still suboptimal, we need to further fine-tune the CodeSteerLLM on pairs of trajectories labeled with preferences. Here we use rule-based scores to assign preferences. Figure 2 illustrates our framework for sampling DPO guidance pairs in a multi-turn setting. The main challenge is sampling and selecting guidance pairs that exhibit clear performance differences across various turns while minimizing the number of samples to conserve resources. We use a tree structure where each node represents a guidance, with a branching factor of 2 or 3. To compare guidance pairs from the same parent node, we calculate their Performance Scores using the following equation:

$$\text{Score}_i = \begin{cases} 15 - i & \text{ending turn/correct,} \\ -i & \text{ending turn/incorrect,} \\ \frac{1}{|C(i)|} \sum_{j \in C(i)} \text{Score}_j & \text{otherwise.} \end{cases} \tag{1}$$

Here, Score_i represents the score for a node at turn i, where i is the current turn number, and $C(i)$ is the set of child nodes of node i. If the current turn is the final one, Score_i is set to $15 - i$ for correct answers and $-i$ for incorrect ones. This incentivizes CodeSteerLLM to achieve correct answers in the fewest turns possible. For non-final turns, Score_i is calculated as the average of its child nodes' scores. This ensures that each non-terminal turn's score reflects the average performance of its potential subsequent actions, i.e., the expectation.

DPO data is collected from guidance pairs within the same parent node at each level that have a score difference greater than 2. To prevent reward hacking (Skalse et al., 2022)—where CodeSteerLLM might bypass exploration and return incorrect answers quickly (e.g., preferring a score of –2 over –5)—we include only pairs where at least one guidance has a positive score. To obtain diverse guidance answers, we set the inference temperature to 1.5 for the SFT fine-tuned CodeSteerLLM and use three models fine-tuned at different epochs (6, 8, and 10) to compare their guidance responses for the same parent node.

5. Experiments

Experimental settings We use GPT-4o as the TaskLLM to test 28 seen and 9 unseen tasks, each with 100 samples of varying complexity. The samples for the 28 seen tasks are different from those used to train CodeSteerLLM. Additionally, we evaluate other LLM types to assess CodeSteer's generalizability.

We compare CodeSteer to six training-free and three training-based baselines, with methods 1, 3–6, and 9 originally proposed in Chen et al. (2025).

Training-free Baselines 1) No extra modifications but only input the original question (**Only Question**); 2) Our framework in Sec. 4.1 to synthesize SFT dataset, where GPT-4o works as CodeSteerLLM with extra hints (**Symbolic Agent**); 3) Prompting LLMs to answer with only text with CoT (**All Text + CoT**); 4) Prompting LLMs to first analyze

Table 1: Experimental results on SymBench. Methods with the highest scores are highlighted blue.

Methods	CoT LLMs			Training-free Methods						Training-based Methods			
Task success rate %	*o1*	*DeepSeek R1*	*o1-preview*	*Only Question*	*Symbolic Agent*	*All Text + CoT*	*All Code + CoT*	*AutoGen Conca.*	*Code + Text + Sum.1*	*Code + Text + Sum.2*	*Code/Text Choice*	*Code Interpreter*	*GPT-4o + CodeSteer*
Ave. Norm., Seen	**83.8**	**79.3**	**77.9**	**59.3**	**77.0**	**56.7**	**71.6**	**73.2**	**66.7**	**65.8**	**79.7**	**73.3**	**88.1**
Ave. Norm., Unseen	**79.4**	**69.1**	**65.1**	**34.5**	**67.9**	**37.9**	**63.2**	**59.5**	**51.9**	**51.7**	**72.1**	**61.9**	**81.3**
Ave. Norm., Total	**82.7**	**76.8**	**74.8**	**53.3**	**74.8**	**52.1**	**69.6**	**69.9**	**63.1**	**62.4**	**77.9**	**70.5**	**86.4**
Seen Tasks													
Game 24	80	65	78	17	37	23	11	88	33	43	43	18	93
Path Plan	74	60	56	65	43	44	76	71	66	61	73	54	75
BoxLift	95	92	85	69	58	56	68	20	65	60	73	49	77
BoxNet	45	43	54	37	30	30	1	12	23	21	23	37	29
Blocksworld	100	100	77	43	60	52	32	50	50	48	44	42	52
Date Understanding	87	88	87	90	89	88	72	65	86	84	86	76	87
Web of Lies	100	100	98	96	99	86	91	78	77	80	98	94	98
Logical Deduction	100	98	97	89	93	91	83	82	94	90	94	82	92
Navigation	100	100	100	98	93	95	99	91	96	94	92	98	99
GSM-Hard	79	77	71	78	76	80	83	81	81	78	77	79	77
MATH Geometry	94	91	90	76	73	73	74	73	77	76	76	73	75
MATH Count&Probab.	96	97	95	89	88	87	88	91	86	88	84	89	93
Logical Equation	100	100	100	52	50	52	40	48	30	33	56	71	78
New Operator	44	39	25	42	39	45	39	47	56	38	48	48	40
Pooling	46	40	42	54	46	60	57	55	43	47	40	49	46
Light Puzzles	100	100	92	62	56	56	69	56	92	78	73	95	68
Mahjong	96	98	93	66	77	73	80	94	72	74	96	64	90
Statistical Counting	25	72	78	34	93	32	95	93	93	86	95	89	97
Matrix Transformation	87	100	98	94	96	76	97	97	96	92	97	90	98
Logical Puzzle	88	80	86	48	58	51	41	39	44	50	44	68	70
Cons. Linear Arrange.	74	62	81	82	71	84	60	79	72	71	77	72	86
Pattern Recognition	100	100	100	70	90	44	89	100	56	60	94	100	93
String Insertion	96	49	72	6	100	8	100	100	67	75	100	89	100
Letter Logic Diagram	50	54	28	2	30	0	12	21	8	9	31	8	45
String Deletion&Modifi.	60	37	34	4	90	0	64	37	51	65	85	49	93
String Synthesis	2	0	2	0	20	0	11	0	7	5	16	12	29
Reversi	46	29	28	8	36	15	49	60	20	23	45	23	52
Standard Sudoku	0	0	0	0	98	0	100	94	12	14	100	100	100
Unseen Tasks													
Letters	61	52	49	12	91	11	100	93	84	87	89	89	96
Eight Queen	84	79	64	8	73	0	35	51	40	45	52	44	78
Number Multiply	43	46	28	11	87	8	100	100	68	65	100	75	95
Cryptanalysis	60	21	49	20	15	24	20	13	16	20	27	0	24
String Splitting	96	91	90	28	52	25	48	47	37	35	48	43	56
Combinatorial Calcul.	57	98	35	16	45	60	55	48	70	67	80	57	86
Synthesis Decompo.	57	96	53	52	53	72	71	35	44	38	69	72	66
2048	52	0	37	44	43	40	28	37	25	20	39	49	56
Permut. and Combina.	100	100	100	66	89	48	64	60	40	46	80	75	93

the question with CoT and then output the code answer (**All Code + CoT**); 5) Concatenating the input question with AutoGen's original system prompt in Appendix Section N (**AutoGen Conca.**); 6) Implement a multi-agent framework that first queries LLMs to answer the question with All Text + CoT and All Code + CoT methods, respectively. Then the final solution is obtained by combining and summarizing both versions of the answers by the same LLM but prompted differently (**Code + Text + Sum.1**).

Training-based Baselines 7) Fine-tune Llama-3.1-8B as a summarizer based on the Code + Text + Sum.1 method using SFT on correct summary data (**Code + Text + Sum.2**); 8) We fine-tune Llama-3.1-8B as a one-step evaluator to choose between text or code generation (**Code/Text Choice**); 9) OpenAI GPT Code Interpreter with the original input question (**Code Interpreter**). Method 7 and 8 are fine-tuned on the same data number and task types as CodeSteer.

Comparison with CoT LLMs We also compare with the current best models: OpenAI o1 and o1-preview (Jaech et al., 2024) and DeepSeek R1 (Guo et al., 2025). These models enhance reasoning and planning by using textual search, reflection, and exploration during answer generation. However, our analysis shows that these CoT LLMs have not yet integrated code-based symbolic computing to further improve their performance.

Evaluations Answers are evaluated using predefined rules, with GPT-4o assisting in adjusting formats as needed. Beyond the Code Interpreter method, some approaches have the LLM output code as the final answer. We extract and execute this code using predefined algorithms to obtain the final result or facilitate further reasoning. To prevent infinite loops, code execution is limited to 30 seconds. If this limit is exceeded, the task is marked as failed or returns errors for subsequent turns. We utilize success rate as the metric for each task. To compare each method, we calculate the Average Normalized Score over all the tested tasks by the following equation:

$$\text{AveNorm}_j = \frac{1}{N}\sum_{i=1}^{N}\frac{s_{ij}}{\max(s_i)} \tag{2}$$

where AveNorm_j is the Average Normalized Score for method j, s_{ij} is the score of method j for task i, $\max(s_i)$ is the maximum score for task i, N is the total number of tasks. This equation normalizes each score relative to the maximum score in the respective task, and then averages the normalized scores over all tasks. We use the normalized metric to better compare the relative performance among methods and prevent any single task from disproportionately influencing the overall evaluation. As shown in Appendix Sec I, to ensure the robustness of our conclusions against changes in the evaluation metric, we recalculate the average score without normalization. From the result in Appendix Table 7, we can observe that for both seen and unseen tasks, our method can still outperform all main baselines obviously, even more on unseen tasks.

Apart from the task performance, in later sections we also discuss the costs of token lengths and runtime for each method.

5.1. Overall Better Performance

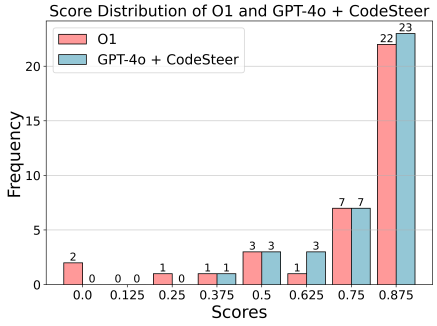

Figure 3: Normalized score distribution of CodeSteer+GPT-4o and o1 in 37 SymBench tasks.

Table 1 presents the full results of all methods on SymBench, including individual task scores and the Average Normalized Score. The key findings are:

1) CodeSteer maintains similar relative performance on seen and unseen tasks, indicating no overfitting.

2) Augmenting GPT-4o with CodeSteer significantly boosts its performance, raising the Ave. Norm. Total Score from 53.3 to 86.4—outperforming all 9 baselines (best baseline: Code/Text Choice at 77.9).

3) GPT-4o + CodeSteer surpasses o1 (82.7), R1 (76.8), and o1-preview (74.8), highlighting the importance of integrating symbolic computing into LLMs. Figure 3 compares the score distribution of GPT-4o + CodeSteer and o1, showing that CodeSteer reduces instances of extremely low scores (near 0), demonstrating its robustness to varied tasks.

4) Compared to other training-based methods (Code + Text + Sum.2 and Code/Text Choice) with the same data number and tasks, CodeSteer's better performance validates the framework's effectiveness (further discussed in Sec. 6).

5.2. Scalability and Generalizability

To assess the impact of symbolic computing, Fig. 4 tracks the performance of five methods across four tasks of increasing complexity. As critical task-specific properties escalate, o1, o1-preview, and GPT-4o fail in highly complex cases, while symbolic-augmented methods (CodeSteer, Code Interpreter) sustain performance. Notably, CodeSteer proves more robust across tasks than Code Interpreter.

In our study, CodeSteerLLM is fine-tuned on synthesized datasets where TaskLLM is always GPT-4o. To assess its

Table 2: Experimental results of Claude-3-5-sonnet-20241022, Mistral-Large, and GPT-3.5 with or without augmented CodeSteer. Methods with the higher scores of the same model are highlighted blue.

Methods	Claude	Claude + CodeSteer	Mistral	Mistral + CodeSteer	GPT-3.5	GPT-3.5 + CodeSteer
Combinatorial Calcu.	48	66	25	34	12	29
Eight Queen	4	87	60	41	0	16
Reversi	0	45	0	33	0	32
Cons. Linear Arran.	73	90	47	48	25	9
Standard Sudoku	0	100	0	100	0	95
Ave. Norm. Score	**29.1**	**92.0**	**31.0**	**59.8**	**8.6**	**42.3**

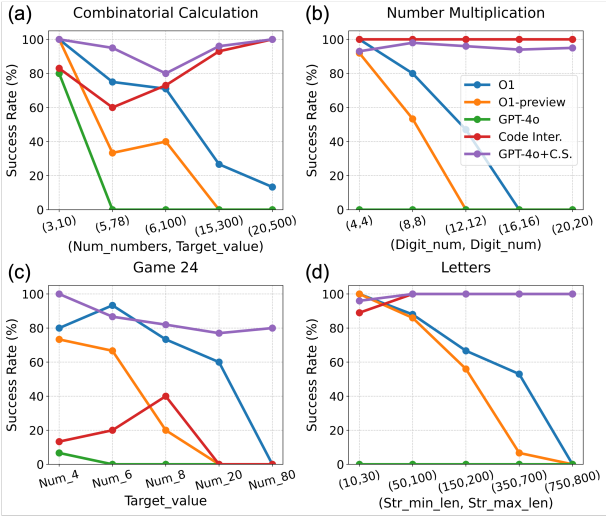

Figure 4: Method performance across four representative tasks as task complexity increases from left to right on the x-axis controlled by value scales. C.S. and Inter. represent CodeSteer and Interpreter.

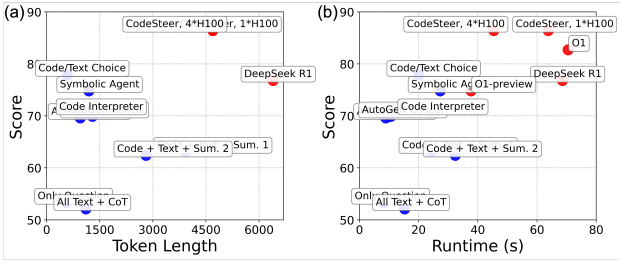

Figure 5: Score vs. token and runtime costs for each method, highlighting CodeSteer, R1, o1, and o1-preview in red. We display CodeSteer results separately for inferences using single or four H100 GPUs. Specific values are in Table 8.

transferability and generalizability, we test it with three popular models: Claude-3-5-Sonnet, Mistral-Large, and GPT-3.5-Turbo. We evaluate them on five representative tasks based on GPT-4o's results in Table 1: two where text outperforms code and three where code is superior. CodeSteer has shown apparent effects when guiding GPT-4o on these tasks. The results in Table 2 confirm that CodeSteer generalizes well across other LLMs types. This is expected, as its core mechanisms—code/text guidance and dynamic adaptation—are essential to all general-purpose LLMs. Notably, we observe that CodeSteer is particularly effective when applied to stronger LLMs, such as Claude. This is likely because more powerful models possess superior self-reflection capabilities and can generate complex code with greater precision. Thus, they benefit more from CodeSteer's additional structured guidance, unlocking their full potential.

5.3. Cost of Tokens and Runtime

Figure 5 shows Score versus Token Length (including input and output tokens) and Score versus Runtime (covering both LLM inference and code execution) for all methods. Complete data is provided in Appendix Table 8. Token counts include only those used by TaskLLM, excluding small and open-source models fine-tuned on Llama-3.1-8B. For the o1 and o1-preview models, only runtime is plotted since their thinking chains are unavailable. While achieving superior performance, CodeSteer uses more tokens than baseline methods due to its multi-turn generations. Most of these tokens are consumed by multiple interaction turns that ultimately fail. CoT LLM R1 consumes more tokens than CodeSteer due to the inefficient textual iteration.

In terms of runtime, CodeSteer is faster than o1 and R1 while delivering better performance. Additionally, since most of CodeSteer's runtime comes from the inference of the 8B CodeSteerLLM on our workstation, hardware and system optimizations can significantly reduce it. For example, running CodeSteerLLM on four H100 GPUs instead of one decreases the average runtime from 63.8 to 45.4 seconds. CoT LLMs consume excessive runtime and tokens due to their extensive and often redundant reasoning chains. Textual iteration is inherently inefficient for search. Appendix Sec. L shows examples of text answers of R1 and GPT-4o, in which both models attempt to find the cor-

rect equation for the Game 24 task by listing all possible combinations, leading to uncontrolled iterations and endless generation. This highlights the importance of symbolic computing through code generation.

5.4. o1 + CodeSteer

Here we test whether CodeSteer can improve the performance of reasoning models like o1. As shown in Table 3, the performance of o1 will improve notably when augmented with CodeSteer on 5 randomly chosen unseen tasks, further verifying the effectiveness of CodeSteer.

5.5. Comparison with Heuristic-based Methods

To better evaluate CodeSteer and provide deeper comparisons, we include three prompt-based baselines. **Few-Shot**: Uses five example-based prompts to guide the TaskLLM in mimicking the 'code/text' switching reasoning process. **Code-First-Rule**: A rule-based approach where the TaskLLM is prompted to use code for the first three rounds (with increasing complexity) and then switch to text-based reasoning. **Code-First-Agent**: Employs GPT-4o as the CodeSteerLLM to guide the TaskLLM using the same code-first-then-text strategy as in Code-First-Rule. As shown in Appendix Table 9, the three prompt-based methods perform significantly worse than CodeSteer, underscoring the effectiveness of training with our synthesized data. Upon analyzing failure cases, we identify two main reasons:

1) CodeSteerLLM's guidance often includes problem-specific coding knowledge (e.g., suggesting A* or DFS) and how to formalize the problem, which purely prompt-based methods struggle to capture.

2) Switching between code and text can be advantageous, as later code generations can build on insights from prior textual reasoning. For instance, in Path Plan, a text-generated trajectory may be partially correct; subsequent code can refine it directly, reducing the search space.

6. Ablation Studies

The CodeSteer framework comprises SFT and DPO dataset synthesis, CodeSteerLLM fine-tuning, a symbolic checker, and a self-answer checker. Here we do the ablation studies on these components and their related modifications. The added experimental results are shown in Table 4 with the whole result table of 37 SymBench tasks in Append Sec. M.

DPO Effects In Table 4, 1.CodeSteer outperforms 2.WO DPO, showing the effectiveness of the DPO process.

SFT Data Augmentation As discussed in Sec. 4.1, we do the data augmentation of the last two turns in each trajectory to prevent multi-turn gradient cancellation. In Table 4, 2.WO DPO achieves higher score than 3.WO DPO WO Data Augment., which means this extra attention on the last two turns does enhance the SFT process.

Symbolic and Self-answer Checkers We evaluate the effects of the Symbolic and Self-answer Checker in two parts: **1) Dataset Synthesis Efficiency:** Comparing Group 6 with Groups 7 and 8 in Table 4 shows that integrating these two checkers increases the Symbolic Agent's success rates, thereby enhancing the efficiency of the dataset synthesis process. **2) CodeSteer Performance:** Comparing Group 1 with Groups 4 and 5 demonstrates that augmenting with these two checkers improves CodeSteer's final performance.

Multi-turn Guidance CodeSteer uses a multi-turn interaction strategy with TaskLLM. In contrast, the Code/Text Choice method in Table 1 relies on single-step guidance and performs worse than CodeSteer. This demonstrates that the multi-turn design enhances guidance effectiveness, aligning with the common intuition that the best methods for many tasks emerge from iterative "executing and exploring" processes accompanied with dynamic adaptation.

Guide Not Summarizer CodeSteer primarily serves as the guidance generator for TaskLLM rather than directly generating answers, summarizing, or selecting among multiple answers. This design choice accounts for the limitations of the open-source LLM we use compared to the more capable closed-source LLM that supports TaskLLM. By focusing on guidance, CodeSteer reduces task complexity and data space requirements. The Code + Text + Sum.2 approach in Table 1 attempts to fine-tune an answer summarizer using the same data volume but fails, highlighting that summarization imposes a significant burden on Llama-3.1-8B due to the unique characteristics of each task.

7. Related Work

Code Generation and Symbolic Computing in LLM Tasks LLMs are widely used for general agent tasks, such as interacting with softwares and websites (Zhou et al., 2023c; Hao et al., 2024a;b; Xu et al., 2024), planning robot actions (Chen et al., 2024d; Ahn et al., 2022), and inferring with logic (Suzgun et al., 2022). Literally, many test tasks in previous works can be solved with direct coding (Suzgun & Kalai, 2024; Gao et al., 2023). Some recent works also further extend the applications of coding into tasks involving commonsense reasoning and semantic analysis (Li et al., 2023; Weir et al., 2024). Most of previous works mainly utilize text (Yao et al., 2024; Ahn et al., 2022; Lin et al., 2023) or code (Liang et al., 2022; Bairi et al., 2024; Zhou et al., 2023a) as the only output modality. Chen et al. (2025) highlights the importance of smartly switching between code and text generation in LLMs but notes current methods have clear drawbacks.

LLM Self-reflection and CoT Models LLM-generated

Table 3: Per-task success rate (%) for the text-only baseline (o1) versus o1 augmented with CodeSteer. For each task, the higher score is highlighted in blue.

Task success rate %	Cryptanalysis	Synthesis Decomposition	2048	Eight Queens	Combinatorial Calculation
o1	**60**	**57**	**52**	**84**	**57**
o1 + CodeSteer	**73**	**94**	**72**	**97**	**95**

Table 4: Ablation studies on CodeSteer. WO DPO: CodeSteer with SFT but without DPO fine-tuning. WO DPO WO Data Augment: Same as WO DPO, but without data augmentation in the last two turns. Agent represents the Symbolic Agent.

Methods Task success rate %	1.Code Steer	2.WO DPO	3.WO DPO WO Data Augment.	4.WO Symbolic Checker	5.WO Self-answer Checker	6. Agent	7.Agent WO Symbolic Checker	8.Agent WO Self-answer Checker
Ave. Norm., Seen	**88.1**	**80.0**	**79.7**	**80.1**	**78.5**	**77.0**	**71.9**	**70.1**
Ave. Norm., Unseen	**81.3**	**76.2**	**70.9**	**68.6**	**64.2**	**67.9**	**62.0**	**57.4**
Ave. Norm., Total	**86.4**	**79.1**	**77.6**	**77.3**	**75.0**	**74.8**	**69.5**	**67.0**

feedback via self-evaluation can improve performance on a variety of tasks (Yang et al., 2022; Welleck et al., 2022; Madaan et al., 2023). The OpenAI o1 (Jaech et al., 2024) and DeepSeek R1 (Guo et al., 2025) models demonstrate the potential of agentic LLMs that use Chain-of-Thought (CoT) text generation to explore and self-reflect, enhancing reasoning and planning. However, they lack symbolic computing and code generation capabilities, leading to weaker performance on complex symbolic tasks and consuming substantial tokens and time (Chen et al., 2024a; Ai et al., 2025).

LLM Fine-tuning with Multi-step SFT and DPO SFT (Chen et al., 2024e) and DPO (Rafailov et al., 2024) are extensively implemented for LLM fine-tuning. To enhance LLM's capability in multi-step agent tasks, these methods are further modified with multi-step goals and rewards (Zhou et al., 2024b; Zhai et al., 2024; Zhang et al., 2024). LLM self-generated data have become increasingly important for model improvement when combined with search algorithms and rejection sampling (Zhou et al., 2023b; Guan et al., 2025).

8. Discussion

Our work underlines the significance of augmenting LLM reasoning and planning capabilities with symbolic computing and shows great potentials of steering large models for smarter code/text generation with specialized small models. We introduce novel modifications to dataset synthesis and fine-tuning (SFT/DPO) to support a multi-turn guidance framework, which has proven effective. Unlike CoT LLMs like OpenAI o1 and DeepSeek R1, which rely solely on textual reasoning for exploration, symbolic computing offers greater efficiency, robustness, and scalability. Since coding is a core LLM capability, generating symbolic tools via code writing preserves generalization across tasks.

Limitations We note that CodeSteer encounters failures under the following conditions, insisting the further research to overcome these bottlenecks.

1) Insufficient Capability of TaskLLM: In some cases, the capabilities of the TaskLLM—whether through coding or textual reasoning—are not sufficient to solve the given problem.

2) Suboptimal Code Generation: The generated code may not use the most efficient method, which can lead to timeouts. For example, as shown in Fig. 4c, CodeSteer's success rate decreases when the target values increase, due to the exponential growth in search complexity.

3) Lack of Robustness to Task Complexity: CodeSteer is not yet robust enough across tasks with varying complexity. As shown in Fig. 4a, performance drops in medium-complexity samples. In these cases, CodeSteer sometimes selects textual reasoning over coding and ends up producing incorrect answers.

Acknowledgments

This work was supported by ONR under Award N00014-22-1-2478 and MIT-IBM Watson AI Lab. However, this article solely reflects the opinions and conclusions of its authors.

Impact Statement

This paper aims to advance the field of Foundation Models. Steering the generation from language models has the great potential to improve safety and performance to better align with human preferences. Any such work is inherently a

double-edged sword; the same techniques used to generate samples from a harmless distribution of text could, with a single sign change, be repurposed for generating samples from a harmful distribution of text. Our method improves language model capability by integrating symbolic computing, which may also be misused for harmful purposes.

Overall, we believe the potential positive social benefits of our work in evaluation and steering language model output towards desired target distributions outweigh the potential negatives stemming primarily from misuse.

References

Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Ahn, M., Brohan, A., Brown, N., Chebotar, Y., Cortes, O., David, B., Finn, C., Gopalakrishnan, K., Hausman, K., Herzog, A., et al. Do as i can, not as i say: Grounding language in robotic affordances. *arXiv preprint arXiv:2204.01691*, 2022.

Ai, M., Wei, T., Chen, Y., Zeng, Z., Zhao, R., Varatkar, G., Rouhani, B. D., Tang, X., Tong, H., and He, J. Resmoe: Space-efficient compression of mixture of experts llms via residual restoration. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.1*, KDD '25, pp. 1–12. ACM, July 2025. doi: 10.1145/3690624.3709196. URL http://dx.doi.org/10.1145/3690624.3709196.

Bairi, R., Sonwane, A., Kanade, A., Iyer, A., Parthasarathy, S., Rajamani, S., Ashok, B., and Shet, S. Codeplan: Repository-level coding using llms and planning. *Proceedings of the ACM on Software Engineering*, 1(FSE): 675–698, 2024.

Chen, W., Ma, X., Wang, X., and Cohen, W. W. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *arXiv preprint arXiv:2211.12588*, 2022.

Chen, X., Xu, J., Liang, T., He, Z., Pang, J., Yu, D., Song, L., Liu, Q., Zhou, M., Zhang, Z., et al. Do not think that much for 2+ 3=? on the overthinking of o1-like llms. *arXiv preprint arXiv:2412.21187*, 2024a.

Chen, Y., Arkin, J., Dawson, C., Zhang, Y., Roy, N., and Fan, C. Autotamp: Autoregressive task and motion planning with llms as translators and checkers. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 6695–6702. IEEE, 2024b.

Chen, Y., Arkin, J., Hao, Y., Zhang, Y., Roy, N., and Fan, C. PRompt optimization in multi-step tasks (PROMST): Integrating human feedback and heuristic-based sampling. In Al-Onaizan, Y., Bansal, M., and Chen, Y.-N. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 3859–3920, Miami, Florida, USA, November 2024c. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.226. URL https://aclanthology.org/2024.emnlp-main.226/.

Chen, Y., Arkin, J., Zhang, Y., Roy, N., and Fan, C. Scalable multi-robot collaboration with large language models: Centralized or decentralized systems? In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 4311–4317. IEEE, 2024d.

Chen, Y., Jhamtani, H., Sharma, S., Fan, C., and Wang, C. Steering large language models between code execution and textual reasoning, 2025. URL https://arxiv.org/abs/2410.03524.

Chen, Z., Deng, Y., Yuan, H., Ji, K., and Gu, Q. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*, 2024e.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. Pal: Program-aided language models. In *International Conference on Machine Learning*, pp. 10764–10799. PMLR, 2023.

Guan, X., Zhang, L. L., Liu, Y., Shang, N., Sun, Y., Zhu, Y., Yang, F., and Yang, M. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

Gui, J., Liu, Y., Cheng, J., Gu, X., Liu, X., Wang, H., Dong, Y., Tang, J., and Huang, M. Logicgame: Benchmarking rule-based reasoning abilities of large language models. *arXiv preprint arXiv:2408.15778*, 2024.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Hao, Y., Chen, Y., Zhang, Y., and Fan, C. Large language models can plan your travels rigorously with formal verification tools. *arXiv preprint arXiv:2404.11891*, 2024a.

Hao, Y., Zhang, Y., and Fan, C. Planning anything with rigor: General-purpose zero-shot planning with llm-based formalized programming. *arXiv preprint arXiv:2410.12112*, 2024b.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Li, C., Liang, J., Zeng, A., Chen, X., Hausman, K., Sadigh, D., Levine, S., Fei-Fei, L., Xia, F., and Ichter, B. Chain of code: Reasoning with a language model-augmented code emulator. *arXiv preprint arXiv:2312.04474*, 2023.

Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., and Zeng, A. Code as policies: Language model programs for embodied control. *arXiv preprint arXiv:2209.07753*, 2022.

Lin, K., Agia, C., Migimatsu, T., Pavone, M., and Bohg, J. Text2motion: From natural language instructions to feasible plans. *Autonomous Robots*, 47(8):1345–1365, 2023.

Madaan, A., Zhou, S., Alon, U., Yang, Y., and Neubig, G. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*, 2022.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36, 2024.

Skalse, J., Howe, N., Krasheninnikov, D., and Krueger, D. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.

Suzgun, M. and Kalai, A. T. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *arXiv preprint arXiv:2401.12954*, 2024.

Suzgun, M., Scales, N., Schärli, N., Gehrmann, S., Tay, Y., Chung, H. W., Chowdhery, A., Le, Q. V., Chi, E. H., Zhou, D., et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

Valmeekam, K., Olmo, A., Sreedharan, S., and Kambhampati, S. Large language models still can't plan (a benchmark for llms on planning and reasoning about change). In *NeurIPS 2022 Foundation Models for Decision Making Workshop*, 2022.

Valmeekam, K., Marquez, M., Olmo, A., Sreedharan, S., and Kambhampati, S. Planbench: An extensible benchmark for evaluating large language models on planning and reasoning about change. *Advances in Neural Information Processing Systems*, 36, 2024.

Wang, J., Wang, J., Athiwaratkun, B., Zhang, C., and Zou, J. Mixture-of-agents enhances large language model capabilities. *arXiv preprint arXiv:2406.04692*, 2024.

Weir, N., Khalifa, M., Qiu, L., Weller, O., and Clark, P. Learning to reason via program generation, emulation, and search. *arXiv preprint arXiv:2405.16337*, 2024.

Welleck, S., Lu, X., West, P., Brahman, F., Shen, T., Khashabi, D., and Choi, Y. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*, 2022.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Zhang, S., Zhu, E., Li, B., Jiang, L., Zhang, X., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Xu, T., Chen, L., Wu, D.-J., Chen, Y., Zhang, Z., Yao, X., Xie, Z., Chen, Y., Liu, S., Qian, B., et al. Crab: Cross-environment agent benchmark for multimodal language model agents. *arXiv preprint arXiv:2407.01511*, 2024.

Yang, K., Tian, Y., Peng, N., and Klein, D. Re3: Generating longer stories with recursive reprompting and revision. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 4393–4479, 2022.

Yang, Y., Xiong, S., Payani, A., Shareghi, E., and Fekri, F. Can llms reason in the wild with programs? *arXiv preprint arXiv:2406.13764*, 2024.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36, 2024.

Zhai, Y., Bai, H., Lin, Z., Pan, J., Tong, S., Zhou, Y., Suhr, A., Xie, S., LeCun, Y., Ma, Y., et al. Fine-tuning large

vision-language models as decision-making agents via reinforcement learning. *arXiv preprint arXiv:2405.10292*, 2024.

Zhang, X., Du, C., Pang, T., Liu, Q., Gao, W., and Lin, M. Chain of preference optimization: Improving chain-of-thought reasoning in llms. *arXiv preprint arXiv:2406.09136*, 2024.

Zhou, A., Wang, K., Lu, Z., Shi, W., Luo, S., Qin, Z., Lu, S., Jia, A., Song, L., Zhan, M., et al. Solving challenging math word problems using gpt-4 code interpreter with code-based self-verification. *arXiv preprint arXiv:2308.07921*, 2023a.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023b.

Zhou, L., Schellaert, W., Martínez-Plumed, F., Moros-Daval, Y., Ferri, C., and Hernández-Orallo, J. Larger and more instructable language models become less reliable. *Nature*, pp. 1–8, 2024a.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Bisk, Y., Fried, D., Alon, U., et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023c.

Zhou, Y., Zanette, A., Pan, J., Levine, S., and Kumar, A. Archer: Training language model agents via hierarchical multi-turn rl. *arXiv preprint arXiv:2402.19446*, 2024b.

Appendices: CodeSteer: Symbolic-Augmented Language Models via Code/Text Guidance

A. Impacts of task types, task complexities, and LLM capabilities on code/text choices

The phenomenon and challenges of steering LLM code/text generation are first proposed by Chen et al. (2025). Here we discuss these phenomenon in details for the motivation of our work. Fig 6 presents two typical examples of the recently popular topics of '9.11' and '9.9' numerical comparison and 'r' letter count in 'strawberry', that the ChatGPT of GPT-4o makes mistakes by direct textual reasoning but easily solves the problem after prompted to use code. Meanwhile, Fig 7 displays the example that GPT-4o makes mistakes to solve the question by code generation but partially solve the question by textual reasoning. The above two examples show that whether code or text is simpler highly depends on the task types and LLM own capabilities and characteristics.

The OpenAI GPT-4o Code Interpreter is trained to steer LLM code/text generation. However, the study of Chen et al. (2025) finds many limitations of this method. In Fig 8, they observe an intriguing property of GPT Code Interpreter: its decision to use code depends on the complexity of the task, as shown in Fig 8. GPT-4o Code Interpreter chooses to handle simple Number Multiplying questions with text and complex questions with code, resulting in correct answers. However, it fails in medium-difficulty questions since it tends to be overconfident and chooses to answer the question via textual reasoning, which sometimes is wrong. Hence, whether to implement symbolic computing depends on task complexities even for the same type of the task.

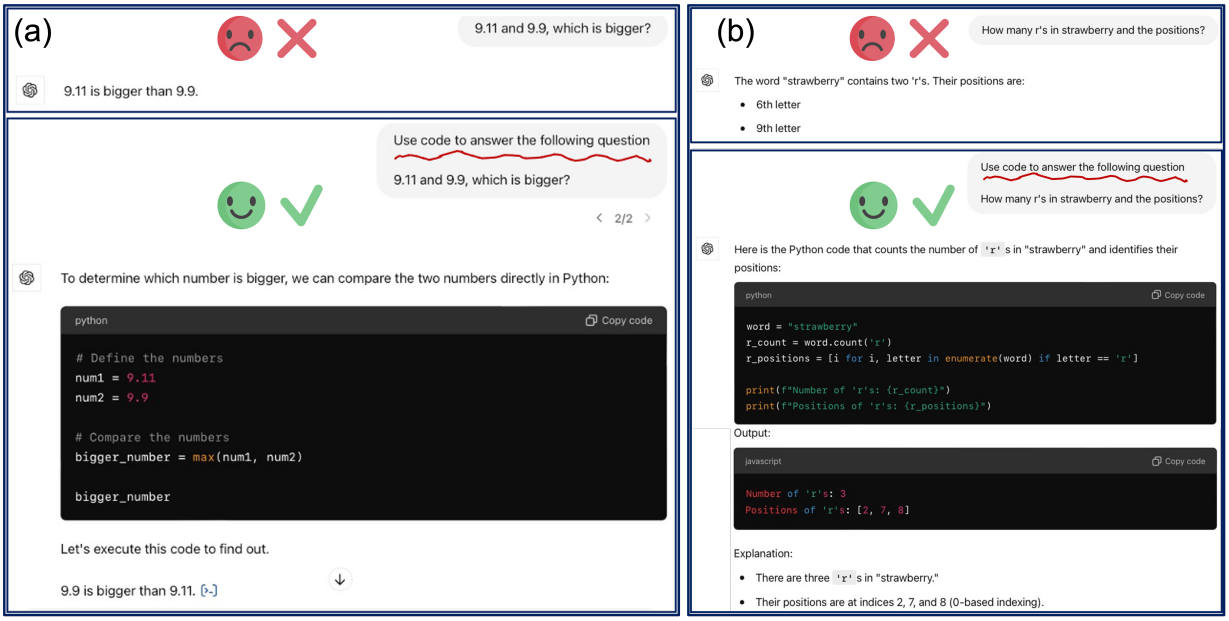

Figure 6: The cases that GPT-4o makes simple mistakes by direct textual reasoning but can reliably solve the problem with prompted to use code.

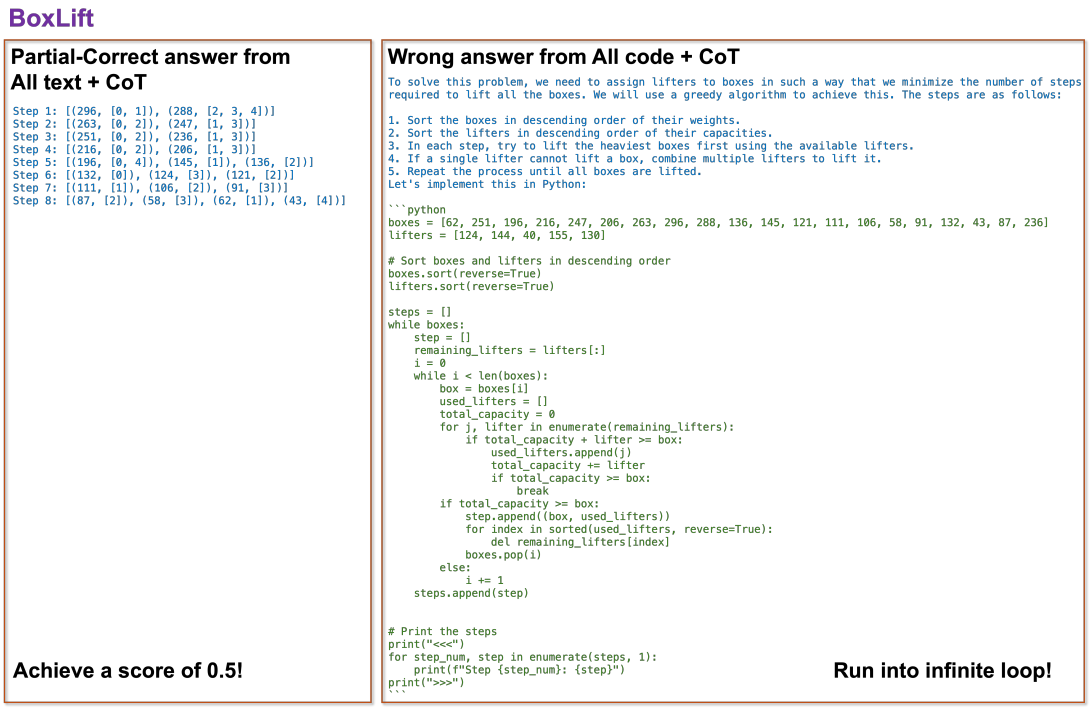

Figure 7: Representative answers of BoxLift task. The left figure is the partially correct answer of GPT-4o with All Text + CoT method. The right figure is the wrong code answer from All Code + CoT method. The text and code parts are colored in blue and green, respectively. The All Code + CoT method generates the wrong code that runs into an infinite loop.

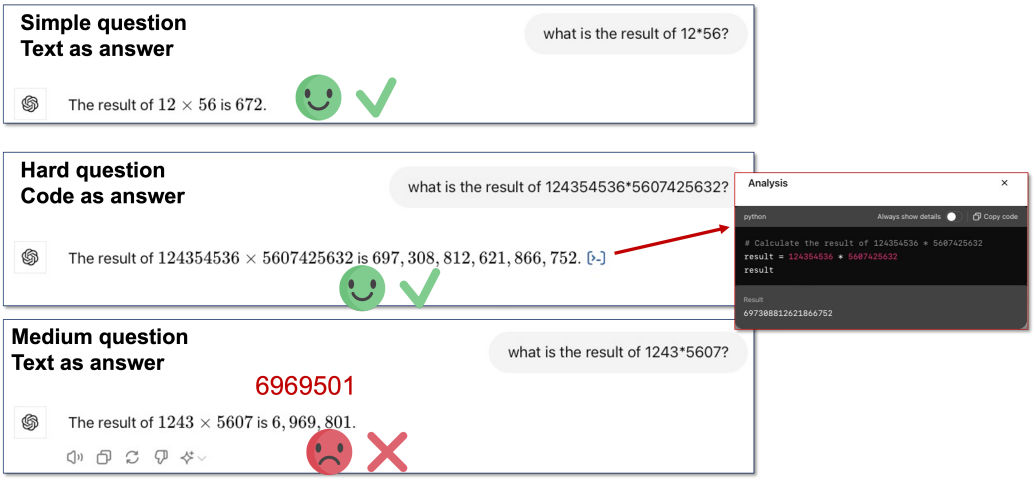

Figure 8: GPT-4o Code Interpreter tends to handle simple Number Multiplying tasks with text and complex tasks with code. However, it often fails with medium-difficulty questions, where it is overconfident and chooses not to use code when needed.

B. Varied code versions of the same LLM

Game 24

Correct code answer of GPT-4o

To solve the problem …<omitted text>…
Let's write a Python script to perform these steps.

```python
# filename: find_24.py
from itertools import permutations, product


def find_24(numbers):
    ops = ['+', '-', '*', '/']
    for num_perm in permutations(numbers):
        for op_perm in product(ops, repeat=3):
            # Generate all possible ways to parenthesize the expression
            expressions = [
                f"({num_perm[0]} {op_perm[0]} {num_perm[1]}) {op_perm[1]} ({num_perm[2]} {op_perm[2]} {num_perm[3]})",
                f"(({num_perm[0]} {op_perm[0]} {num_perm[1]}) {op_perm[1]} {num_perm[2]}) {op_perm[2]} {num_perm[3]}",
                f"({num_perm[0]} {op_perm[0]} ({num_perm[1]} {op_perm[1]} {num_perm[2]})) {op_perm[2]} {num_perm[3]}",
                f"{num_perm[0]} {op_perm[0]} (({num_perm[1]} {op_perm[1]} {num_perm[2]}) {op_perm[2]} {num_perm[3]})",
                f"{num_perm[0]} {op_perm[0]} ({num_perm[1]} {op_perm[1]} ({num_perm[2]} {op_perm[2]} {num_perm[3]}))"
            ]
            for expr in expressions:
                try:
                    if abs(eval(expr) - 24) < 1e-6:  # Check if the result is 24
                        return f"<<<{expr} = 24>>>"
                except ZeroDivisionError:
                    continue
    return "No solution found."


numbers = [3, 5, 8, 13]
print(find_24(numbers))
```

…<omitted text>…

Wrong code answer of GPT-4o

To solve the problem of using the numbers 3, 5, 8, and 13 to obtain 24, we need to find a combination of arithmetic operations that will result in 24. Each number must be used exactly once.
Let's break down the approach:

Identify possible operations: We can use addition, subtraction, multiplication, and division.

Try different combinations: We need to experiment with different groupings and operations to see which combination yields 24.

Ensure all numbers are used: The solution must incorporate all four numbers exactly once.

After testing various combinations, we find that the following equation works:

(3×8)−(13−5)=24

Now, let's write the Python code to compute and display this result:

```python
# Define the numbers
a = 3
b = 5
c = 8
d = 13

# Calculate the result using the identified equation
result = (a * c) - (d - b)

# Print the result in the required format
print("<<<", result, ">>>")
```

Figure 9: Representative code answers of Game 24 task. The left figure is the correct code of GPT-4o with extra AutoGen prompt in Appendix Sec. N for guiding code/text choices. The right figure is the wrong code after prompting GPT-4o to answer with code 'Think of an algorithm to solve the task and implement it in python'. The text and code parts are colored in blue and green, respectively. In both cases, GPT-4o is prompted to solve this task with code. The only difference is the guiding prompts. However, GPT-4o answers with different types of codes, with or without efficient symbolic computing. This phenomenon shows that LLM code generation is unstable under varied prompts, tasks, and LLM types.

C. Description of SymBench tasks

Here we describe the 37 testing tasks. They require strong symbolic, mathematical, logical, geometrical, scientific, and commonsense reasoning capabilities. The first 14 tasks originate from Chen et al. (2025), while the last 23 are from Gui et al. (2024). Note that both these two previous works do not release the full question datasets and codes for these 37 tasks. The released question dataset in Gui et al. (2024) only contains 8 or 16 questions for each task. Hence, we develop codes to automatically synthesize the questions for each task with tunable complexities. Both our developed codes and question datasets are released.

Number Multiplying This task involves querying LLMs to compute the product among integers. It represents a classic problem that LLMs are not able to solve through pure textual reasoning.

Game 24 This task involves querying LLMs to use a given set of integers to generate an equation that evaluates to 24. This task is tested in previous work Tree-of-Thought (Yao et al., 2024).

Path Plan This task involves querying LLMs to plan the robot trajectory waypoints based on human task instructions and environments. This task originates from AutoTAMP (Chen et al., 2024b).

Letters This task involves querying LLMs to count the total number of specific letters in a long word and specify their positions. An example question can be 'How many r's in the word strawberry and what are their positions?'. This task has recently gained significant attention because current LLMs struggle to perform it effectively and accurately.

BoxLift This task involves coordinating robots of various types to lift boxes of different sizes and weights. Each robot has a specific lifting capacity and can collaborate with others to lift a single box. A box can only be lifted if the combined lifting capacity of the robots exceeds the box's weight. The objective is to lift all the boxes in the minimum number of time steps. This task originates from Scalable-Robots (Chen et al., 2024d).

BoxNet This task involves coordinating robot arms to move colored boxes (squares) into corresponding colored goal locations (circles) in the fewest time steps. Each robot arm is assigned and restricted to a cell indicated by the dotted lines. The arms have two possible actions: (1) move a box within their cell to a neighboring cell, or (2) move a box within their cell to a goal location within the same cell. The objective is to ensure all boxes are placed in their matching goal locations efficiently. This task originates from Scalable-Robots (Chen et al., 2024d).

Blocksworld In Blocksworld, the objective is to stack a set of blocks (brown) according to a specific order. The robot can perform four actions: (1) pick up a block, (2) unstack a block from the top of another block, (3) put down a block, (4) stack a block on top of another block. A robot can only pick up, unstack, or stack a block if it is clear, that is, the block has no other blocks on top and is not currently being held. This task originates from PlanBench (Valmeekam et al., 2024).

Date Understanding Given a small set of sentences referring a specific date, the task involves querying LLMs to answer a provided question based on the information in these sentences (e.g., 'The concert was scheduled for 06/01/1943, but was delayed by one day to today. What was the date yesterday in MM/DD/YYYY?'). This task originates from BIG-Bench-Hard (Suzgun et al., 2022).

Web of Lies This task involves querying LLMs to determine the truth value of a random Boolean function presented as a natural-language word problem. This task originates from BIG-Bench-Hard (Suzgun et al., 2022).

Logical Deduction This task involves querying LLMs to deduce the order of a sequence of objects using clues and information about their spacial relationships and placements. This task originates from BIG-Bench-Hard (Suzgun et al., 2022).

Navigate This task involves querying LLMs to determine whether the agent would return to its initial starting point after following a series of navigation steps. This task originates from BIG-Bench-Hard (Suzgun et al., 2022).

GSM-Hard (Gao et al., 2023) This is the more challenging version of GSM8K (Cobbe et al., 2021) math reasoning dataset, where the numbers in the original questions of GSM8K are replaced with larger, less common values.

MATH-Geometry This is the math reasoning dataset from MATH dataset (Hendrycks et al., 2021), with specific focus on geometry questions.

MATH-Count&Probability This is the math reasoning dataset from MATH dataset (Hendrycks et al., 2021), with specific focus on counting and probability questions.

The following 23 tasks originate from LogicGame (Gui et al., 2024).

Logical Equation The task is to assign a specific numeric value to each letter from a given set, using a predefined range of numbers and a set of inequalities. Each letter corresponds to a unique number, and the relationships between the letters are defined by mathematical equations or constraints.

New Operator This task introduces custom mathematical operations involving two numbers, defined with unique formulas. The goal is to use the given definitions of these operations to compute the result of a specific expression.

Pooling This task involves applying a pooling operation on a numerical $N \times N$ grid. The pooling operation uses an $n \times n$ sliding window ($n < N$) that moves across the grid from left to right and top to bottom. The results from each window are then arranged based on their positions to create a new output matrix.

Light Puzzles In this task, you are given an $n \times n$ grid representing a network of lights, where a lit light is represented by "1" and an unlit light by "0". Several buttons control the state of these lights by turning them on or off in certain positions. The state of each light can be affected by multiple buttons. The task is to follow a series of button presses and determine the final state of the grid.

Mahjong Given an initial set of letter cards, in each turn, a new card is added and one card is removed. Some effects may happen when specific combinations of the cards appear after introducing the new card. A result is determined based on these specific conditions. The goal is to determine a result based on a series of turns

Statistical Counting Calculate the total score of a string by scanning it from left to right, where consecutive identical letters earn points (for example, two or more consecutive A's add 1 point, B's add 2 points, etc.). The task is to start with a score of 0 and return the final summing value.

Matrix Transformation Rotate a given matrix of characters based on given instruction (e.g., 90 degrees clockwise), preserving each character's position relative to others in the transformed output. The input matrix can be of any size and contain any character.

Logical Puzzle The task involves querying LLMs to select a specified number of different values from a grid of numbers, ensuring that certain mathematical constraints (sum or product) are satisfied for selected numbers for each row and column.

Constrained Linear Arrangement In a two-player card game, the task is to deduce your opponent's moves based on the game's rules, your played cards, and the announced results of each turn. Each card can only be used once, and the game follows specific interaction rules between different card types, where certain cards can defeat, be defeated by, or draw with others according to predefined relationships.

Pattern Recognition The task involves querying LLMs to find all squares in a character matrix where each square consists of identical characters and has a side length of at least 3.

String Insertion The task is to transform a string by scanning it from left to right and inserting specific characters after certain character patterns (e.g., each pattern WXYZ requires inserting W immediately after it occurs). All operations are performed simultaneously on the original string.

Letter Logic Diagram The task is to complete an incomplete grid by selecting from a list of letters, where each row and column must contain each letter exactly once, and all cells on the minor diagonal (top-right to bottom-left) must contain the same letter. Some cells are already filled in as constraints.

String Deletion and Modification The task is to transform a string by repeatedly applying a set of ordered string manipulation rules until no more changes are possible, where each rule modifies the string based on specific patterns or conditions present in the current string state. For example, a modification rule can be "If the string ends with 'ba', replace it with 'ab'."

String Synthesis Given an initial set of blocks and a set of synthesis rules that combine different types of blocks, the task is to determine the final block(s) after repeatedly applying these rules in order until no more combinations are possible.

Reversi In this game similar to Reversi, players take turns placing pieces on an $n \times n$ grid. After placing a piece, any of the opponent's pieces located between two of the player's pieces (in the same row, column, or diagonal) will be flipped. The task is to determine the state of the board after rounds, starting from a given configuration.

Standard Sudoku Given a partially filled Sudoku grid, the task is to fill the remaining empty cells with numbers between

1 and 9, ensuring that no number repeats in the same row, column, or 3×3 subgrid.

Eight Queen Given a grid with some queens already placed, the task is to place the remaining queens such that no two queens share the same row, column, or diagonal, while avoiding positions with obstacles in the grid.

Cryptanalysis In this task, you are provided with a combination lock consisting of numbers and letters, where neither the numbers nor the letters repeat. Using a series of guesses and feedback, the goal is to deduce the correct password based on the given conditions.

String Splitting A dismantling engineer has old machines and can obtain machine parts through a set of predefined methods. By continuously cycling through these methods in a specific order, the engineer dismantles machines or combines parts to create new components, and the task is to determine the total number of parts and remaining machines after all possible cycles.

Combinatoral Calculation Given a set of integers, the goal is to use arithmetic operations (addition, subtraction, multiplication, division) and parentheses to arrange the numbers in such a way that the final result matches a specified target value. Each number must be used exactly once, and the order of the numbers cannot be changed.

Synthesis Decomposition A farmer grows various crops and can exchange them for agricultural products. Using a set of methods, he can trade specific combinations of crops for products, following a cyclic pattern until no further exchanges are possible. The goal is to determine the synthesis result for each round.

2048 Similarly to the 2048 game, in a grid, numbers representing powers of 2 can move in any direction, combining when they encounter a matching number to form the next power of 2. Given a starting position and a sequence of movements, the goal is to determine the resulting grid after executing the moves.

Permutation and Combination Given a set of objects with specific positioning constraints, the task is to determine the correct arrangement of the objects on a shelf. Each object must be placed in a position according to the rules provided, ensuring that the conditions on adjacency, order, and specific positions are met. For example, a rule about adjacency could be 'Book A must be adjacent to book I'.

Table 5: The evaluated capabilities of all tasks, classified as Execution, Planning, and Reasoning tasks.

Categories	Tasks	Mathematics	Spatial Reasoning	Logical Reasoning	Order Reasoning	Optimization	Search
Execution	Number Multiplying	✓	✗	✗	✗	✗	✗
	New operator	✓	✗	✗	✗	✗	✗
	Pooling	✓	✓	✗	✗	✗	✗
	Light Puzzles	✗	✓	✗	✗	✗	✗
	Mahjong	✗	✗	✗	✓	✗	✗
	Statistical Counting	✓	✗	✗	✓	✗	✗
	Matrix Transform.	✗	✓	✗	✗	✗	✗
	Pattern Recognition	✗	✓	✗	✗	✗	✓
	String Insertion	✗	✗	✓	✓	✗	✓
	String Deletion &Modi.	✗	✗	✓	✓	✗	✓
	String Synthesis	✗	✗	✓	✓	✗	✓
	Reversi	✗	✓	✗	✗	✗	✗
	String Splitting	✗	✗	✓	✓	✗	✓
	Synthesis Decomposition	✗	✗	✓	✓	✗	✓
	2048	✓	✓	✓	✗	✗	✗
Planning	Game 24	✓	✗	✗	✓	✓	✗
	Path Plan	✗	✓	✗	✓	✗	✓
	Letters	✗	✓	✗	✗	✗	✓
	BoxLift	✗	✗	✓	✗	✓	✗
	BoxNet	✗	✗	✓	✗	✓	✗
	Blocksworld	✗	✓	✓	✗	✓	✗
	Logical Equation	✓	✗	✓	✗	✗	✓
	Logic Puzzle	✓	✓	✗	✗	✗	✓
	Const. Linear Arrange.	✗	✗	✓	✗	✗	✗
	Letter Logic Diagram	✗	✓	✓	✗	✗	✗
	Standard Sudoku	✓	✓	✗	✗	✗	✓
	Eight Queen	✗	✓	✗	✗	✗	✗
	Cryptanalysis	✗	✗	✓	✗	✗	✗
	Combinatorial Calculation	✓	✗	✗	✗	✓	✗
	Permutation and Combina.	✗	✓	✓	✓	✗	✗
Reasoning	Date Understanding	✗	✗	✓	✗	✗	✗
	Web of Lies	✗	✗	✓	✗	✗	✗
	Logical Deduction	✗	✗	✓	✗	✗	✗
	Navigate	✗	✓	✗	✓	✗	✗
	GSM-Hard	✓	✗	✓	✗	✗	✗
	MATH-Geometry	✓	✓	✗	✗	✗	✗
	MATH-Count&Probability	✓	✗	✓	✗	✗	✓

D. Prompt for CodeSteerLLM

The input prompts of CodeSteerLLM follow a multi-turn dialogue, i.e., previous turns of prompts and responses will be included as history prompts for following generation of response guidance. Since we set the maximum turns of guidance to be 5 for each task, the total addition of prompt and output lengths of CodeSteerLLM does not surpass maximum context window 8k. The formats for the first turn of prompt and following turns of prompts are as follows. Note that 'The summary of generated code complexity is: {code_complexity_summary}' is not included if the generated answer by TaskLLM does not have code.

Turn 1 prompt to CodeSteerLLM
You are guiding another TaskLLM to solve a task. You will be presented with a task that can potentially be solved using either pure textual reasoning or coding. Your goal is to determine which method will be most effective for solving the task. Follow these steps:
Respond with the chosen approach but not the solution. You can choose between the following options:
- If you choose coding, explain the reasons and respond the final returned guidance with the format `<<<guidance prompt content>>>` in the end of your response.
- If you choose textual reasoning, explain the reasons and respond the final returned guidance with the format `<<<guidance prompt content>>>` in the end of your response.
Now, here is the task:

Following Turns of prompts to CodeSteerLLM
The response from TaskLLM is: {response}
The feedback from the checking agent is: {check_result}
The summary of generated code complexity is: {code_complexity_summary}
The final returned guidance prompt should be of the format `<<<guidance prompt content>>>`.

E. Prompt for Self-answer Checker

Prompt for Self-answer Checker
Given the following question and the answer from other LLMs, write a python code block to check the correctness of the answer. Try to generate the code to check the correctness of the answer. Try your best to check whether the answer satisfy all the constraints of the given question. If the answer is correct, return the text "Correct". If the answer is incorrect, return the reason why the answer is wrong, like what condition or constraint is not satisfied.
Question: {question}
Answer: {answer}

F. Code for Symbolic Checker

The following code checks the factors of iteration, search, numeric, permutations, and combinations in the answered code by TaskLLM and returns the summary of code complexity and the complexity score. We directly return the summary of code complexity as 'code_complexity_summary' to CodeSteerLLM for further guidance. If the complexity score less than 2.0, the returned 'code_complexity_summary' concatenates with 'The generated code may not be complex enough to carry out symbolic computing for solving the task.'

```
def analyze_code_and_explain(code_string):
    """Analyzes Python code for computational approaches and provides explanation."""
    import ast, re
    try:
        tree = ast.parse(code_string)
        analysis = {'search': False, 'symbolic': False, 'numeric': False, 'expl': [], 'score': 0}

        # Check patterns
        for node in ast.walk(tree):
            if isinstance(node, (ast.For, ast.While)):
                analysis['search'] = True
                analysis['expl'].append("Contains loops for systematic search")
            elif isinstance(node, (ast.Import, ast.ImportFrom)) and 'itertools' in node.names[0].name:
                analysis['search'] = True
                analysis['expl'].append("Uses itertools for combinatorial search")
            elif isinstance(node, ast.BinOp):
                analysis['numeric'] = True
                analysis['expl'].append("Contains direct mathematical operations")
                analysis['score'] += 0.25

        # Check symbolic patterns
        if any(x in code_string for x in ['eval(', 'exec(', 'f"', "f'", '.format(']):
            analysis['symbolic'] = True
            analysis['expl'].append("Uses string operations for expression handling")

        # Calculate complexity
        analysis['score'] += sum(len(re.findall(pattern, code_string))
                                 for pattern in ['for|while', 'if|elif|else', 'try|except'])

        # Generate explanation
        result = []
        for comp_type, detected in [('SEARCHING', analysis['search']),
                                    ('SYMBOLIC', analysis['symbolic']),
                                    ('NUMERICAL', analysis['numeric'])]:
            if detected:
                result.append(f"\n{comp_type} APPROACH detected:")
                result.extend([f"- {exp}" for exp in analysis['expl']
                               if any(x in exp.lower() for x in [comp_type.lower()[:6], 'operation', 'loop'])])

        return "\n".join(result + [f"\nComplexity score: {analysis['score']}"]), analysis['score']

    except Exception as e:
        return f"Invalid Python code: {str(e)}", 0
```

Figure 10: Code for checking the symbolic factors of the generated code by TaskLLM.

G. Synthesized dataset number of each task for SFT and DPO

Table 6: Synthesized dataset number of each task for SFT and DPO fine-tuning processes.

Dataset number	SFT success trajectory number	DPO pair number
Game 24	792	320
Path Plan	442	215
BoxLift	345	163
BoxNet	330	186
Blocksworld	406	248
Date Understanding	497	238
Web of Lies	492	204
Logical Deduction	489	241
Navigation	503	170
GSM-Hard	332	125
MATH Geometry	342	115
MATH Count&Prob.	346	127
Logical Equation	396	213
New Operator	394	189
Pooling	404	187
Light Puzzles	406	259
Mahjong	421	230
Statistical Counting	402	223
Matrix Transform.	391	214
Logical Puzzle	454	148
Constrained Linear Arrangement	432	155
Pattern Recognition	414	135
String Insertion	409	128
Letter Logic Diagram	500	226
String deletion&Modification	504	230
String Synthesis	397	185
Reversi	403	194
Standard Sudoku	400	212
Total	12043	5480

H. Parameter and hardware settings of SFT/DPO fine-tuning and inference processes

We utilize four H100 80GB GPUs for full-parameter fine-tuning of the Llama-3.1-8B models. The model is trained for 10 epochs in the SFT stage and 6 epochs in the DPO stage. The learning rate is set to 1×10^{-5} for SFT and 5×10^{-6} for DPO. We use a batch size of 4 for training. In DPO, the loss function follows the standard sigmoid loss (Rafailov et al., 2024), with the hyperparameter β set to 0.1.

In most cases, we perform the inference of CodeSteerLLM using a single H100 80GB GPU. However, to analyze the impact of hardware configurations on CodeSteer runtime, as shown in Fig. 5, we also conduct inference using four H100 GPUs for comparison.

For the generation of guidance answers in the DPO dataset creation, we utilize three different SFT fine-tuned Llama-3.1-8B models, trained for 6, 8, and 10 epochs, respectively. For each question and stage, we query all three models and compare their generated guidance answers.

I. Score comparison of different methods without normalization across tasks

Table 7: Average (non-normalized) score (%) for each method. The best score in every row is highlighted in blue.

Avg. Score	o1	DeepSeekR1	Symbolic Agent	Code/Text Choice	Code Interpreter	GPT-4o + Codesteer
Seen	**73.7**	**70.4**	**67.5**	**70.0**	**64.9**	**76.1**
Unseen	**67.8**	**64.8**	**60.9**	**64.9**	**56.0**	**72.2**
Total	**72.3**	**69.0**	**65.9**	**68.8**	**62.8**	**75.2**

J. Score-cost table for each method

Table 8: Score-cost table for each method.

Average Norm.	Average score (↑)	Average token length (↓)	Average runtime (s) (↓)
Baseline Methods			
Only Question	53.3	566.1	8.2
Symbolic Agent	74.8	1192.5	27.3
All Text + CoT	52.1	1110.7	15.3
All Code + CoT	69.6	949.8	8.9
AutoGen Conca.	69.9	1295.9	10.6
Code + Text + Sum. 1	63.1	3931.6	24.2
Code + Text + Sum. 2	62.4	2808.6	32.4
Code/Text Choice	77.9	587.4	20.1
Code Interpreter	70.5	1175.9	23.8
CoT LLMs			
DeepSeek R1	76.8	6396.6	68.6
o1	82.7	N/A	70.5
o1-preview	74.8	N/A	37.7
Proposed Methods			
CodeSteer, 1*H100	**86.4**	4693.3	63.8
CodeSteer, 4*H100	**86.4**	4693.3	45.4

K. Comparison with heuristic-based methods

Table 9: Task success rate for CodeSteer versus three heuristic-based methods. The highest score for each task is shown in blue.

Task Success Rate %	CodeSteer	Few-Shot	Code-First-Rule	Code-First-Agent
Game 24	**93**	28	68	76
Path Plan	**75**	54	59	57
Eight Queen	**78**	47	62	73
Combinatorial Calculation	**86**	58	47	59
2048	**56**	49	40	48

L. Example text answer of DeepSeek R1 and GPT-4o in Game 24

Okay, let's see. I need to use the numbers 1, 3, 13, and 13 to make 24. Each number has to be used exactly once. Hmm. Let's start by listing out all the possible operations and combinations.
First, the numbers are 1, 3, 13, 13. Wait, there are two 13s. Oh, but the problem says each number has to be used once. But since both 13s are the same, maybe that gives some flexibility. But I need to use both of them, right?
So let's think. Maybe I can combine the 13s in some way. Like 13 divided by 13 is 1. But then I already have a 1. Hmm. So if I do 13/13 =1, then I have 1 and 1 and 3. But then how to get 24 from 1, 1, 3. That might not help. Maybe subtract them? 13 -13 =0, which probably isn't useful here.
Alternatively, adding them: 13 +13 =26. Then I have 26, 3, 1. How to get from 26,3,1 to 24. 26 - (3 -1) =24? Let's check: 26 -2 =24. But that would require 26 - (3-1) =24. So that would be (13+13) - (3 -1) =24. Let me verify: 13+13 is 26. 3-1 is 2. 26-2 is 24. Yes! But wait, have I used all the numbers? Let's check. 13,13,3,1. Yes. Each is used once. So the equation would be (13 +13) - (3 -1) =24. Let me write that with parentheses correctly: ((13 +13) - (3 -1)) =24. Alternatively, it's (13+13)-(3-1) =24. That works. Let me check again: 13+13=26, 3-1=2, 26-2=24. Yep. So that's a possible solution. Let me see if there are other ways.
Alternatively, maybe using multiplication. Let's see: 3 * something. For example, 3*8=24. So if I can make 8 with the remaining numbers 1,13,13. Let's see. 13 - (13/1) =0. Not helpful. 13 -1=12. 12 +13=25. Not 8. Hmm. Alternatively, 13 - (13 -1) =1. Still not 8.
What about 13 divided by 13 is 1, then 3-1=2. Then 1*2=2. No, that's not enough. Maybe 3*(1 + ...). Let me think. 3*(8) =24. So how to get 8. Let's see. Maybe 13 - (13 - (1 + ...)). Hmm. Not sure. Let's try another approach.
What if I do 13 * (something) to get close to 24. 13*2=26. 26 - 2=24. But where does 2 come from? 3-1=2. So 13*2 -2=24. Wait, but how to structure that. Let's see: 13*(3-1) - (13 - ?). Wait, maybe not. Let's see. If I use 3-1=2, then 13*2=26. Then 26 - (13 - x) where x is the remaining number. But the remaining number is already used? Wait, perhaps not. Let's re-examine. If I use 13*(3-1), that uses 13,3,1. Then subtract the other 13? So 13*(3-1) -13 =26-13=13. Not 24. Hmm.
Alternatively, 13*(something) divided by something. Let me think. 13 * (3) is 39. Then 39 - (13 +1) =25. Not 24. Close. 13*3 -13 -1 =39-14=25. Not enough. Hmm.
Wait, another idea: 13 + 13 =26. Then subtract 3 and add 1. 26-3+1=24. So that's 26-3+1=24. But how to arrange that. The expression would be (13+13) -3 +1. That uses all numbers: 13,13,3,1. Each once. So (13+13) -3 +1 =24. Let's compute: 26-3 is 23, plus 1 is 24. Yes. So that works. So another possible solution is 13+13-3+1=24. But the problem is that sometimes the order of operations might require parentheses. Let's check: 13+13 is 26, minus 3 is 23, plus 1 is 24. So the order is left to right, so no parentheses needed. So 13+13-3+1=24. But does that fit the required format? The previous examples used parentheses, but maybe it's allowed. Let me check the previous answers. For example, the first input's answer was ((10-9)*(11+13)) =24, which uses parentheses. But in another example, the answer was (2-(6-(10+18))) which simplifies to 2 -6 +28 =24. So maybe parentheses can be adjusted as needed. So in this case, 13+13-3+1=24. Let me confirm. 13+13 is 26, minus 3 is 23, plus 1 is 24. Correct. That uses all numbers once. So that's a valid solution.
But wait, maybe the problem expects some grouping. Let me see. If I write (13+13) -3 +1, but that's the same as 13+13-3+1. Alternatively, ((13 +13) - (3 -1)) which we thought of earlier. Both are correct. Which one is better? Let's see. Let's compute the second version: (13+13) - (3-1) =26 -2=24. So that uses all numbers. So either way, there are two possible solutions here.
But the problem says "each number has to be used in the equation". So as long as all four numbers are used once, either solution is okay. Let's check both.
First solution: 13 +13 -3 +1 =24. Numbers used: 13,13,3,1. Correct.
Second solution: (13+13) - (3-1) =24. Numbers used: 13,13,3,1. Correct.
So both are valid. Which one to choose? Maybe the second one uses parentheses, which is more in line with the examples given, where parentheses are used. For example, the first example was ((10-9)*(11+13)), so grouping operations. The second example: ((4*11)-(10+10)). So perhaps the answer should be written with parentheses to show the grouping. But since both are correct, either could be acceptable. However, perhaps the second one is more explicit about the order of operations, even though in the first case, left to right works. Let's verify if the first solution is correct without parentheses. Let me see: 13+13-3+1. Since addition and subtraction have the same precedence, it's evaluated left to right. So 26 -3 is 23, plus 1 is 24. Correct. So that's a valid expression. So the answer could be either (13+13-3+1) or ((13+13)-(3-1)), or similar variations. Let's check if there are other possibilities.
Another approach: 3*(13 - (13/1)). Let's compute that. 13/1 is 13. 13-13=0. 3*0=0. No good. Alternatively, (13 -1) * (3 - (13/13)). 13/13 is 1. 3-1=2. 12 *2=24. Oh, wait, that's a possible solution. Let's break it down:
(13 -1) is 12.
(3 - (13/13)): 13/13=1, so 3-1=2.
Then 12 *2 =24. So the equation is (13 -1) * (3 - (13/13)) =24. Let's check the numbers: 13,1,3,13. All used once. Correct. So that's another valid solution. Let me verify:
13-1=12, 13/13=1, 3-1=2, 12*2=24. Yes. So that's another way. This might be a better solution because it uses multiplication and division, which might be necessary for some other combinations. So this could be the intended answer. Let me check again:
(13 - 1) * (3 - (13 / 13)) =12 * (3-1)=12*2=24. Correct. So that uses all numbers once. So this is another valid solution.
So now I have three possible solutions:
1. 13 +13 -3 +1 =24
2. (13 +13) - (3 -1) =24
3. (13 -1) * (3 - (13/13)) =24
All of them are valid. Which one should I choose? The third one involves multiplication and division, which is similar to some of the examples. For instance, the third example: ((5 - (13/13)) *6) = (5-1)*6=24. So the third solution here is similar in structure. So maybe that's the expected answer. Let's see. Let me check the numbers again for solution 3: 13,1,3,13. All used once. Yes. Each number is used exactly once. So yes. So this is valid.
So the answer could be either of these. But since the examples tend to use grouping, perhaps the third solution is the one to go with. Let me write it properly:
((13 -1) * (3 - (13/13))) =24
Breaking it down:
13/13 =1
3 -1=2
13-1=12
12 *2=24. Correct.
Alternatively, another way to write it: (13-1)*(3 - (13/13)) =24. That's correct.
Alternatively, maybe even without the inner parentheses: (13-1)*(3 -13/13). But division has higher precedence, so it's the same as (3 - (13/13)).
So the answer is ((13 -1) * (3 - (13 / 13))) =24. Which is the third solution.
Alternatively, another possible solution: 13*(3 -1) - (13 - something). Wait, no. Let's see. 13*(3-1) =26. 26 -2=24. But how to get 2 from 13 and 1. 13 - (1) is 12, which doesn't help. Wait, 13-1 is 12. 26 - (12) =14. No. Not helpful. So that approach might not work.
So the third solution seems solid. Let me check if there are any other possible combinations.
Another idea: 13 * ((3 -1) + (13 /13)). Let's compute that. 13/13=1. 3-1=2. 2+1=3. 13*3=39. No, not 24. Hmm.
Alternatively, (13 + 1) * (3 - (13/13)). 13+1=14. 3-1=2. 14*2=28. Not 24.
Hmm. What about (13 - 3) * (13 -1)/something? Let's see. 13-3=10. 13-1=12. 10*12=120. 120 divided by something. 120/5=24. But we don't have a 5. So that doesn't help.
Alternatively, (13 * 3) - (13 +1). 39 -14=25. Not 24. Close.
Another idea: 13 * ((3 +1) - (13/13)). Let's compute. 13/13=1. 3+1=4. 4-1=3. 13*3=39. No.
Alternatively, 3 * (13 - (13 -1)/something). Not sure. Let's see: 13-1=12. 13-12=1. 3*1=3. Not helpful.
Alternatively, (13 *1) + (13 -3) =13 +10=23. No. Not 24.
Wait, what if we do 13 * (1 + (3 - (13 /13)))? Let's break it down. 13/13=1. 3-1=2. 1+2=3. 13*3=39. No. Not helpful.
Hmm. Let's go back to the earlier solutions. The third solution seems the most plausible, using multiplication and division, which are common in the examples. For example, the first example uses multiplication, the second subtraction and addition, the third uses division and multiplication, etc. So the third solution fits that pattern.
So the answer is ((13 -1) * (3 - (13 /13))) =24. Let me confirm again:
13/13=1
3 -1=2
13-1=12
12 *2=24. Correct.
Yes. That's a valid equation using all four numbers once. Each number is used exactly once. 13 is used in (13-1) and in (13/13), 3 is used in (3 - ...), and 1 is from the initial 1. Wait, wait. Wait, the input numbers are 1,3,13,13. So in this solution:
- The first 13 is used in (13 -1)
- The 1 is used in (13 -1)
- The second 13 is used in (13/13)
- The 3 is used in (3 - ...)
Yes, each number is used once. The two 13s are both used, each in different parts. So that's correct.
Alternatively, if the problem had said that the two 13s are considered the same number and can't be used twice, but the problem says each number should be used once. Since the input is 1,3,13,13, then each of the four numbers (including both 13s) must be used once. So the solution is valid.
Therefore, the answer should be ((13 -1) * (3 - (13 /13))) =24.

Figure 11: Example text answer of R1 in the task Game 24. R1 searches possible answers with the continuous back-and-forth textual reasoning process. This search process still fails in the end.

Game 24, GPT-4o text answer

```
To solve for 24 using the numbers 2, 3, 5, and
12, we can try different combinations of
arithmetic operations. Let's explore the
possibilities:

1. \( 12 - 2 = 10 \)
2. \( 10 - 5 = 5 \)
3. \( 5 \times 3 = 15 \) (This doesn't work)

Let's try another combination:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work either)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 3 = 21 \)
3. \( 21 + 2 = 23 \) (This doesn't work)

Let's try a different approach:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 3 = 9 \)
2. \( 9 \times 2 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Let's try another combination:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Let's try another combination:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)


Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)


Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)


Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)


Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)
```

```
Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
1. \( 12 - 5 = 7 \)
2. \( 7 \times 2 = 14 \)
3. \( 14 + 3 = 17 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 + 3 = 13 \)
3. \( 13 + 5 = 18 \) (This doesn't work)

Another attempt:
1. \( 12 \div 3 = 4 \)
2. \( 4 \times 5 = 20 \)
3. \( 20 + 2 = 22 \) (This doesn't work)

Another attempt:
1. \( 12 - 2 = 10 \)
2. \( 10 \div 5 = 2 \)
3. \( 2 \times 3 = 6 \) (This doesn't work)

Another attempt:
1. \( 12 \div 2 = 6 \)
2. \( 6 \times 3 = 18 \)
3. \( 18 + 5 = 23 \) (This doesn't work)

Another attempt:
……
```

Figure 12: Example text answer of GPT-4o in the task Game 24. GPT-4o continues the textual reasoning process until reaching the maximum token generation length but never returns the answer.

M. Full experimental results of ablation studies

Table 10: Full experimental results of ablation studies on the components in CodeSteer framework.

Methods Task success rate %	1.Code Steer	2.WO DPO	3.WO DPO WO Data Augment.	4.WO Symbolic Checker	5.WO Self-answer Checker	6. Agent	7.Agent WO Symbolic Checker	8.Agent WO Self-answer Checker
Ave. Norm., Seen	**88.1**	**80.0**	**79.7**	**80.1**	**78.5**	**77.0**	**71.9**	**70.1**
Ave. Norm., Unseen	**81.3**	**76.2**	**70.9**	**68.6**	**64.2**	**67.9**	**62.0**	**57.4**
Ave. Norm., Total	**86.4**	**79.1**	**77.6**	**77.3**	**75.0**	**74.8**	**69.5**	**67.0**
Game 24	93	93	46	62	57	37	41	28
Path Plan	75	76	74	72	74	43	41	29
BoxLift	77	65	76	66	72	58	47	39
BoxNet	29	21	31	13	17	30	24	15
Blocksworld	52	50	50	54	51	60	45	41
Date Understanding	87	83	86	80	83	89	84	92
Web of Lies	98	94	92	95	92	99	95	97
Logical Deduction	92	92	95	91	89	93	91	87
Navigation	99	90	95	85	80	93	94	88
GSM-Hard	77	74	72	79	74	76	73	70
MATH Geometry	75	74	70	71	69	73	68	70
MATH Count&Prob.	93	92	86	84	81	88	85	82
Logical Equation	78	58	56	61	56	50	52	56
New Operator	40	38	40	24	52	39	28	20
Pooling	46	43	51	47	45	46	44	52
Light Puzzles	68	71	52	51	52	56	56	60
Mahjong	90	88	88	92	95	77	85	79
Statistical Counting	97	98	92	95	84	93	90	96
Matrix Transform.	98	100	97	96	95	96	92	96
Logical Puzzle	70	58	56	52	44	58	53	54
Const. Linear Arrange.	86	66	65	76	81	71	64	52
Pattern Recognition	93	96	95	95	93	90	92	100
String Insertion	100	100	100	100	100	100	100	100
Letter Logic Diagram	45	20	35	35	35	30	25	23
String deletion&Modi.	93	88	92	90	88	90	86	76
String Synthesis	29	12	21	30	26	20	12	14
Reversi	52	49	39	52	24	36	28	36
Standard Sudoku	100	100	95	100	100	98	100	100
Letters	96	85	88	87	84	91	79	75
Eight Queen	78	74	72	72	52	73	64	52
Number Multiply	95	90	92	94	95	87	80	74
Cryptanalysis	24	22	15	4	12	15	12	7
String Splitting	56	56	31	43	41	52	42	40
Combinatorial Calculation	86	76	88	65	76	45	60	56
Synthesis Decomposition	66	62	64	44	60	53	56	44
2048	56	56	44	53	44	43	32	40
Permutation and Combina.	93	86	80	92	56	89	82	78

N. System prompt of AutoGen

System prompt of AutoGen (Wu et al., 2023)
You are a helpful AI assistant. Solve tasks using your coding and language skills. In the following cases, suggest python code (in a python coding block) or shell script (in a sh coding block) for the user to execute. 1. When you need to collect info, use the code to output the info you need, for example, browse or search the web, download/read a file, print the content of a webpage or a file, get the current date/time, check the operating system. After sufficient info is printed and the task is ready to be solved based on your language skill, you can solve the task by yourself. 2. When you need to perform some task with code, use the code to perform the task and output the result. Finish the task smartly. Solve the task step by step if you need to. If a plan is not provided, explain your plan first. Be clear which step uses code, and which step uses your language skill. When using code, you must indicate the script type in the code block. The user cannot provide any other feedback or perform any other action beyond executing the code you suggest. The user can't modify your code. So do not suggest incomplete code which requires users to modify. Don't use a code block if it's not intended to be executed by the user. If you want the user to save the code in a file before executing it, put # filename: filename inside the code block as the first line. Don't include multiple code blocks in one response. Do not ask users to copy and paste the result. Instead, use 'print' function for the output when relevant. Check the execution result returned by the user. If the result indicates there is an error, fix the error and output the code again. Suggest the full code instead of partial code or code changes. If the error can't be fixed or if the task is not solved even after the code is executed successfully, analyze the problem, revisit your assumption, collect additional info you need, and think of a different approach to try. When you find an answer, verify the answer carefully. Include verifiable evidence in your response if possible. Reply "TERMINATE" in the end when everything is done.