Code Execution as Grounded Supervision for LLM Reasoning

Dongwon Jung[1] **Wenxuan Zhou** [2] **Muhao Chen**[1]
[1]University of California, Davis, [2]University of Southern California,
{dwojung,muhchen}@ucdavis.edu zhouwenx@usc.edu

Abstract

Training large language models (LLMs) with chain-of-thought (CoT) supervision has proven effective for enhancing their reasoning abilities. However, obtaining reliable and accurate reasoning supervision remains a significant challenge. We propose a scalable method for generating a high-quality CoT supervision dataset by leveraging the determinism of program execution. Unlike existing reasoning dataset generation methods that rely on costly human annotations or error-prone LLM-generated CoT, our approach extracts verifiable, step-by-step reasoning traces from code execution and transforms them into a natural language CoT reasoning. Experiments on reasoning benchmarks across various domains show that our method effectively equips LLMs with transferable reasoning abilities across diverse tasks. Furthermore, the ablation studies validate that our method produces highly accurate reasoning data and reduces overall token length during inference by reducing meaningless repetition and overthinking.[1]

1 Introduction

Large language models (LLMs) have demonstrated strong performance across a range of complex reasoning tasks. A key development in this area is chain-of-thought (CoT) training, which enhances LLMs by encouraging the generation of intermediate reasoning steps before producing a final answer (Chung et al., 2024; Ho et al., 2022; Magister et al., 2022; Li et al., 2023). CoT supervision has proven especially effective in improving the generalization and interpretability of LLMs, and has become a central component in the development of reasoning models (Ye et al., 2025; Muennighoff et al., 2025; Team, 2025; Chang et al., 2025).

Despite its success, obtaining high-quality CoT data at scale remains a major challenge for supervising the reasoning LLMs. Existing CoT datasets are typically constructed in two ways. First, human-annotated CoT examples (Chung et al., 2024; Cobbe et al., 2021) provide high-quality and accurate reasoning guidance but are costly to acquire and non-scalable. Second, many recent efforts rely on bootstrapped CoT data generated by prompting existing LLMs (Magister et al., 2022; Muennighoff et al., 2025). However, these synthetic data often suffer from intermediate reasoning errors, inconsistencies, and lack of grounding (Zheng et al., 2024a; Lyu et al., 2023; Chen et al., 2025). Although these methods verify and filter the CoT data at either the process or outcome level (Zelikman et al., 2022; Lightman et al., 2023; Luo et al., 2024; Li et al., 2025), they still fall short in guaranteeing the correctness of intermediate reasoning steps, undermining the reliability of the supervision signal.

In this work, we propose a scalable method for generating verifiable CoT data to supervise the reasoning process of LLM by leveraging the determinism of program execution. Our core insight is that when problems can be formalized and solved with executable code, the resulting execution traces (Åkerblom et al., 2014) provide inherently correct, step-by-step reasoning aligned with the task. These traces offer a verifiable and error-free alternative to LLM-generated CoTs and can serve as a trustworthy source of supervision.

Specifically, we begin by sourcing open-source Python programs and executing them with a debugger to extract rich execution traces, including intermediate variable values, line-level execution order, and program control flow. Since the resulting raw execution traces lack natural language reasoning structures, we employ LLMs to translate the raw execution traces into fluent, human-readable rationales that resemble natural CoT data, effectively combining the correctness guarantees of ex-

[1]Code and data are available at https://github.com/luka-group/Execution-Grounded-Reasoning

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 24823–24834
November 4-9, 2025

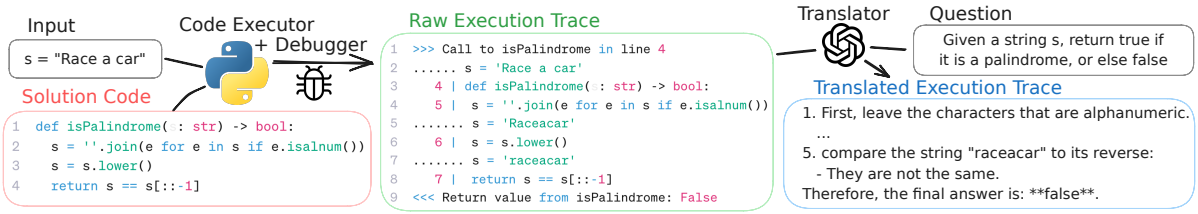

Figure 1: An overview of our method. The translated execution trace is grounded in code execution, making it a reliable and accurate source of reasoning supervision for the LLM.

ecution with the expressive power of LLMs. Our method offers a scalable, annotation-free pipeline for generating high-quality and accurate reasoning supervision.

Experiments show that LLMs trained with our method demonstrate its effectiveness, achieving robust performance across coding, math, and reasoning tasks compared to baseline approaches. Ablation studies further confirm that our method improves data quality and reduces overall token length by mitigating meaningless repetition and overthinking.

2 Method

2.1 Problem Settings

Our goal is to enhance the reasoning capabilities of LLMs by supervising them with accurate and verifiable CoT reasoning traces. Formally, given an input sequence $\mathbf{x} = [x_1, \ldots, x_m]$, an LLM $p_{\boldsymbol{\theta}}$ generates an output sequence $\mathbf{y} = [y_1, \ldots, y_n]$ through a sequence of intermediate reasoning steps $\mathbf{s} = [s_1, \ldots, s_l]$. The overall generation process is defined as:

$$p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{x}) = p_{\boldsymbol{\theta}}(\mathbf{y}|\mathbf{s}, \mathbf{x}) \prod_{t=1}^{l} p_{\boldsymbol{\theta}}(s_t|\mathbf{s}_{<t}, \mathbf{x}),$$

where the model first generates each reasoning step s conditioned on the input $\mathbf{x}$ and previous steps $\mathbf{s}_{<t}$, followed by generating the final answer $\mathbf{y}$ based on the full reasoning trace $\mathbf{s}$ and the original input $\mathbf{x}$.

High-quality CoT data is crucial for enabling strong reasoning performance in LLMs (Lightman et al., 2023). To collect CoT data at scale, existing approaches adopt a *generate-then-filter* paradigm: they first sample CoTs using LLMs and then filter out low-quality ones. Outcome-level filtering typically checks whether the final answer $\mathbf{y}$ matches the ground truth (Xiong et al., 2025), but this can miss flawed reasoning that coincidentally produces correct answers. Process-level filtering is more desirable as it evaluates intermediate reasoning quality, but remains challenging for current LLMs (Zheng et al., 2024a).

In this work, we propose a fundamentally different approach: constructing CoT data from code execution traces, which are inherently step-by-step, accurate, and causally linked to the final outcome, making them a natural source of accurate and verifiable reasoning supervision. In the following sections, we describe how we construct high-quality CoT data, which is then used to fine-tune the LLM via supervised fine-tuning.

2.2 Execution Trace Acquisition

To efficiently obtain reliable and accurate CoTs, we leverage the abundance of coding data, which provide supervision in the form of problem–solution code pairs. These pairs allow us to ground reasoning supervision in executable programs that reflect correct problem-solving logic. Specifically, given a solution code snippet $\mathbf{c}$ and an input $\mathbf{x} = [\mathbf{x}_q; \mathbf{x}_i]$, where $\mathbf{x}_q$ is a natural language problem description and $\mathbf{x}_i$ is a concrete test input (e.g., a specific string or numerical input), we execute $\mathbf{c}$ using an execution tracing tool to obtain the returned answer $\mathbf{y}$ and a detailed execution trace $\mathbf{s}_{\text{trace}}$:

$$\mathbf{y}, \mathbf{s}_{\text{trace}} = \texttt{Code\_Executor}(\mathbf{c}, \mathbf{x}_i).$$

We employ a Python debugging tool called Snoop (Hall, 2024) as the execution tracing tool, which records detailed line-by-line execution signals, including function calls and returns, executed lines of code, and the updated local variable values. An example of a Snoop-generated execution trace is provided in Figure 1.[2] The resulting dataset is denoted as $D_{\text{trace}} = (\mathbf{x}, \mathbf{s}_{\text{trace}}, \mathbf{y})$, which contains verifiably accurate execution trace $\mathbf{s}_{\text{trace}}$ and the correct final output $\mathbf{y}$ for each instance, grounded in a verifiable code executor.

[2] A more illustrative example of an execution trace is provided in Appendix B

2.3 Execution Trace Translation

Although the execution trace $\mathbf{s}_{\text{trace}}$ captures the verifiable, step-by-step problem solving logic, its format differs substantially from natural language CoT reasoning. Therefore, directly fine-tuning on such traces may hinder generalization and risks catastrophic forgetting on other reasoning tasks. To better align the supervision signals with natural language reasoning, we transform the raw trace $\mathbf{s}_{\text{trace}}$ into a natural language CoT $\mathbf{s}_{\text{nl\_trace}} = (s_1, ..., s_m)$ using an LLM as a `Translator`:

$$\mathbf{s}_{\text{nl\_trace}} = \texttt{Translator}(\mathbf{x}, \mathbf{s}_{\text{trace}}).$$

The translator is prompted to emulate a human solving the problem by mentally tracing the code. It is instructed to express each reasoning step in natural language while faithfully reflecting the exact values and logic observed during code execution. This ensures that the output mirrors the precise reasoning behind the program's behavior—grounded in execution, yet phrased as natural, intuitive, step-by-step thinking. The resulting dataset, $D_{\text{nl\_trace}} = (\mathbf{x}, \mathbf{s}_{\text{nl\_trace}}, \mathbf{y})$, provides high-quality CoT traces that are both verifiable and linguistically aligned with typical LLM data, making them ideal for supervised fine-tuning.

3 Experiment

To assess whether supervision from code execution traces genuinely enhances reasoning ability, we conduct comprehensive experiments by comparing our approach with baseline datasets specifically designed to improve the reasoning capabilities of LLMs. In this section, we present the experimentation details and discuss the results.

3.1 Experiment Setup

Data Generation We select PyEdu-R, a subset of data from the Python-Edu (Ben Allal et al., 2024) as the source of supervision dataset. PyEdu-R focuses on STEM-related problems such as logic puzzles, math-related tasks, scientific computation, and system modeling. Since the original data only contains code, we utilize the preprocessed version that contains LLM-generated problem and the input-ouput pairs, made publicly available by Li et al. (2025). We obtain execution traces by running the code on the inputs and then translate these traces using `Qwen3-32B` as the `Translator`. This process yields approximately 15K data instances.[3]

Baselines We compare our method against several baselines designed to enhance LLM reasoning through fine-tuning. The training setup remains the same, with the only difference being the dataset curation process from the source data. The compared baselines are: (1) **No Training**, where the base LLM is evaluated without further fine-tuning; (2) **Code Generation**, where the model is trained to generate solution code $\mathbf{c}$ given a question $\mathbf{x}_{\text{q}}$; (3) **Raw Execution Trace**, where the model learns to generate the raw execution trace $\mathbf{s}_{\text{trace}}$ directly from the code $\mathbf{c}$ and input $\mathbf{x}_{\text{i}}$, bypassing the natural language translation; and (4) **CodeI/O** (Li et al., 2025), where the model is trained on CoT traces $\mathbf{s}_{\text{teacher}}$ produced by a teacher model, followed by a binary output correctness feedback. For consistency, we use the same model employed in our `Translator` as the teacher in this baseline.

Models We conduct SFT on our data using two target models: `Qwen3-4B` and `Qwen3-8B`. We use `Qwen3-32B` as both the translator in our method and the teacher model for the CodeI/O baseline. For both methods, we enable the `enable_thinking=True` option and extract the output after the thinking phase for the translation and the CoT generation results.

Evaluation Benchmark We evaluate our method and the baselines on widely adopted reasoning benchmarks including MATH500 (Lewkowycz et al., 2022), BBH (Suzgun et al., 2022), AGIEval (Zhong et al., 2023), and GPQA (Rein et al., 2024). Additionally, we utilize LiveBench (White et al., 2025), a recent comprehensive benchmark that contains diverse categories of tasks. We focus on the math, coding, and reasoning categories, as our primary goal is to evaluate the reasoning capabilities of the methods. Specifically, we use the 2024-11-25 release, which is currently the most up-to-date version.

3.2 Experiment Results

The experiment results of `Qwen3-4B` is presented in Table 1.[4] Overall, our method outperforms all baselines, demonstrating its effectiveness. The Code Generation and Raw Execution Trace baselines perform poorly across the board. Although both are

[3] We also include more details on data generation in Appendix A.2 and Appendix A.3

[4] The experiment result of `Qwen3-8B` is presented in Table 6

Methods	LiveBench			MATH500	BBH	AGIEval	GPQA	Avg
	Code	Math	Reasoning					
No Training	43.2	59.5	64.2	86.4	75.5	31.6	36.3	56.7
Code	28.5	33.1	51.7	82.4	69.7	31.2	33.9	47.2
Raw Trace	8.3	3.1	38.0	84.8	63.5	30.2	27.0	36.4
CodeI/O	39.3	**62.7**	62.2	**86.6**	81.3	**32.4**	33.9	56.9
Ours	**44.5**	61.0	**65.8**	86.4	**81.4**	**32.4**	**38.1**	**58.5**

Table 1: Experiment results (accuracy) of `Qwen3-4B`. Bolded scores indicate the highest performance.

derived from code-based supervision, they suffer from a misalignment with natural language reasoning formats. In particular, training on raw traces or code generation alone does not equip the model with the ability to produce step-by-step reasoning in natural language, leading to limited transfer and degraded performance—especially in reasoning-heavy tasks.

CodeI/O baseline remains strong on mathematical reasoning but suffers from a significant drop in performance on coding and science domains. This is likely because the reasoning structure of the teacher model, which is primarily optimized for math, is effectively distilled into the student model—potentially at the cost of performance in other domains. In contrast, our method achieves balanced improvements across all reasoning domains.

3.3 Ablation Studies

In this section, we present ablation studies evaluating the quality of our data and its effectiveness in reducing repetition and overthinking during inference.

Better Quality Data. To assess the quality of the data generated by our method, we evaluate the correctness of (1) final output and (2) intermediate reasoning steps. We provide `Qwen3-32B` with the generated solution and the ground truth answer to determine if it reaches the correct answer. For evaluating intermediate reasoning steps, we randomly select 200 samples and use a strong reasoning model, `OpenAI-o3`, to identify any errors within the reasoning process.

Table 2 shows the accuracy of both the final outputs and intermediate reasoning steps for our method and CodeI/O. Our method achieves higher accuracy than CodeI/O on both metrics, with a particularly larger margin in intermediate step accuracy. This demonstrates that our method produces more accurate reasoning steps, leading to the correct final output, as it is grounded in reliable code execution.

Method	Output	Intermediate
CodeI/O	87.3	73.0
Ours	98.3	91.5

Table 2: Comparison of correctness accuracy of final output and intermediate reasoning steps on LiveBench.

Reducing Repetition and Overthinking. To evaluate the generation efficiency of the models trained on our method, we compare token lengths in Table 3. We show that models trained with our approach generate approximately 20% fewer tokens for `Qwen3-4B` and 30% fewer tokens for `Qwen3-8B`, compared to No Training baseline.[5] Moreover, our method substantially reduces instances where the model reaches the maximum token limit due to overthinking or repetitive output. Importantly, this reduction in token length does not compromise performance, as demonstrated in Section 3.2.

3.4 Case Study

To further examine our data, we present two examples of reasoning trace in Table 9 and Table 10. The first example demonstrates a case where the CodeI/O solution produces the correct final output despite an error in an intermediate step—specifically, it incorrectly lists the permutation of "hrf" in step 2—while our solution correctly completes all intermediate steps. The second example shows a case where both the intermediate reasoning and the final output are incorrect in the CodeI/O solution, due to an incorrect formula used at the beginning. In contrast, our solution produces correct calculations throughout, guided by the execution trace.

[5]We include `Qwen3-8B` results in Table 5

Method	Avg Token	Max Token Reached
No Training	8804	123
CodeI/O	7684	91
Ours	7068	72

Table 3: Token length statistics on the LiveBench evaluation using `Qwen3-4B`.

4 Related Works

Reasoning Distillation from Teacher Models A common approach to distill reasoning ability is supervised fine-tuning on reasoning chains from stronger teacher models (Ho et al., 2022; Magister et al., 2022; Li et al., 2023). More recent work leverages test-time scaling to transform instruction-tuned models into Meta Chain-of-Thought (Meta-CoT; Xiang et al. 2025), which first generate thought tokens before solving the problem. Our work is orthogonal to these approaches that leverage test-time scaling, and aims to improve the model's inherent step-by-step reasoning ability.

Training on Code for Reasoning Early studies have shown that LLMs trained on code excel at various reasoning tasks, including commonsense reasoning (Madaan et al., 2022), causal reasoning (Liu et al., 2023, 2024), and mathematical reasoning (Azerbayev et al., 2023; Shao et al., 2024). However, these findings are limited to models heavily pre-trained on code and do not deeply investigate how code semantics and structure influence reasoning abilities.

Recent studies have leveraged code execution traces to enhance reasoning on coding tasks, focusing on applications like vulnerability detection, program repair, and code generation (Ding et al., 2024b; Ni et al., 2024; Ding et al., 2024a). In contrast, our work targets task-agnostic, step-by-step reasoning that extends beyond the code domain.

The most closely related work is by Li et al. (2025), who train models on input–output prediction tasks using CoT rationales. While both approaches leverage CoT for output prediction, our method uses execution traces as grounded supervision, whereas theirs relies solely on binary output correctness.

5 Conclusion

We introduce a novel approach that leverages code execution traces as verifiable and easily scalable supervision for enhancing step-by-step reasoning in LLMs. Our method incorporates both the ensured correctness of execution with the natural CoT reasoning steps to provide LLMs with high-quality reasoning supervision. Experiments across reasoning tasks show that our approach outperforms prior distillation methods, offering a reliable path toward improving LLM reasoning with grounded, annotation-free supervision.

Acknowledgment

We appreciate the reviewers for their insightful comments and suggestions. This work was partly supported by the Amazon Nova Trusted AI Prize, the NSF of the United States Grants ITE 2333736 and OAC 2531126, and the DARPA FoundSci Grant HR00112490370.

Limitations

While our method provides grounded and reliable reasoning supervision, it is inherently limited to tasks that can be expressed and solved via executable code. However, we have proved that our reasoning data transfers well to other domains like math and logical reasoning. Additionally, since the translation relies on an LLM, it may not always be perfect. Nonetheless, we have demonstrated the quality of our data by evaluating both the final outputs and the intermediate reasoning steps.

Ethics Statement

This work follows the ACL Code of Ethics. We believe no potential risk is directly associated with the presented work.

References

Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlin, and Tobias Wrigstad. 2014. Tracing dynamic features in python programs. In *Proceedings of the 11th working conference on mining software repositories*, pages 292–295.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q Jiang, Jia Deng, Stella Biderman, and Sean Welleck. 2023. Llemma: An open language model for mathematics. *arXiv preprint arXiv:2310.10631*.

Loubna Ben Allal, Anton Lozhkov, Guilherme Penedo, Thomas Wolf, and Leandro von Werra. 2024. Smollm-corpus.

Edward Y. Chang, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. 2025. Demystifying long chain-of-thought reasoning in llms. *ArXiv*, abs/2502.03373.

Yanda Chen, Joe Benton, Ansh Radhakrishnan, Jonathan Uesato, Carson Denison, John Schulman, Arushi Somani, Peter Hase, Misha Wagner, Fabien Roger, and 1 others. 2025. Reasoning models don't always say what they think. *arXiv preprint arXiv:2505.05410*.

Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Alex Castro-Ros, Marie Pellat, Kevin Robinson, and 16 others. 2024. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024a. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems*, 37:60275–60308.

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. 2024b. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12.

Alex Hall. 2024. Snoop. https://github.com/alexmojaki/snoop.

Namgyu Ho, Laura Schmid, and Se-Young Yun. 2022. Large language models are reasoning teachers. In *Annual Meeting of the Association for Computational Linguistics*.

Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramasesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, and 1 others. 2022. Solving quantitative reasoning problems with language models. *Advances in Neural Information Processing Systems*, 35:3843–3857.

Junlong Li, Daya Guo, Dejian Yang, Runxin Xu, Yu Wu, and Junxian He. 2025. Codei/o: Condensing reasoning patterns via code input-output prediction. *ArXiv*, abs/2502.07316.

Liunian Harold Li, Jack Hessel, Youngjae Yu, Xiang Ren, Kai-Wei Chang, and Yejin Choi. 2023. Symbolic chain-of-thought distillation: Small models can also "think" step-by-step. In *Annual Meeting of the Association for Computational Linguistics*.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. 2023. Let's verify step by step. In *The Twelfth International Conference on Learning Representations*.

Xiao Liu, Zirui Wu, Xueqing Wu, Pan Lu, Kai-Wei Chang, and Yansong Feng. 2024. Are llms capable of data-based statistical and causal reasoning? benchmarking advanced quantitative reasoning with data. *arXiv preprint arXiv:2402.17644*.

Xiao Liu, Da Yin, Chen Zhang, Yansong Feng, and Dongyan Zhao. 2023. The magic of if: Investigating causal reasoning abilities in large language models of code. *arXiv preprint arXiv:2305.19213*.

Liangchen Luo, Yinxiao Liu, Rosanne Liu, Samrat Phatale, Meiqi Guo, Harsh Lara, Yunxuan Li, Lei Shu, Yun Zhu, Lei Meng, and 1 others. 2024. Improve mathematical reasoning in language models by automated process supervision. *arXiv preprint arXiv:2406.06592*.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. 2023. Faithful chain-of-thought reasoning. In *The 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (IJCNLP-AACL 2023)*.

Aman Madaan, Shuyan Zhou, Uri Alon, Yiming Yang, and Graham Neubig. 2022. Language models of code are few-shot commonsense learners. *arXiv preprint arXiv:2210.07128*.

Lucie Charlotte Magister, Jonathan Mallinson, Jakub Adamek, Eric Malmi, and Aliaksei Severyn. 2022. Teaching small language models to reason. *ArXiv*, abs/2212.08410.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. 2025. s1: Simple test-time scaling. *arXiv preprint arXiv:2501.19393*.

Ansong Ni, Miltiadis Allamanis, Arman Cohan, Yinlin Deng, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2024. Next: teaching large language models to reason about code execution. In *Proceedings of the 41st International Conference on Machine Learning*, pages 37929–37956.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2024. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Y Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, and 1 others. 2022. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*.

NovaSky Team. 2025. Sky-t1: Train your own o1 preview model within $450. https://novasky-ai.github.io/posts/sky-t1. Accessed: 2025-01-09.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, and 1 others. 2025. Livebench: A challenging, contamination-free llm benchmark. In *The Thirteenth International Conference on Learning Representations*.

Violet Xiang, Charlie Snell, Kanishk Gandhi, Alon Albalak, Anikait Singh, Chase Blagden, Duy Phung, Rafael Rafailov, nathan lile, Dakota Mahan, Louis Castricato, Jan-Philipp Franken, Nick Haber, and Chelsea Finn. 2025. Towards system 2 reasoning in llms: Learning how to think with meta chain-of-thought. *ArXiv*, abs/2501.04682.

Wei Xiong, Jiarui Yao, Yuhui Xu, Bo Pang, Lei Wang, Doyen Sahoo, Junnan Li, Nan Jiang, Tong Zhang, Caiming Xiong, and 1 others. 2025. A minimalist approach to llm reasoning: from rejection sampling to reinforce. *arXiv preprint arXiv:2504.11343*.

Yixin Ye, Zhen Huang, Yang Xiao, Ethan Chern, Shijie Xia, and Pengfei Liu. 2025. Limo: Less is more for reasoning. *arXiv preprint arXiv:2502.03387*.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. 2022. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488.

Chujie Zheng, Zhenru Zhang, Beichen Zhang, Runji Lin, Keming Lu, Bowen Yu, Dayiheng Liu, Jingren Zhou, and Junyang Lin. 2024a. Processbench: Identifying process errors in mathematical reasoning. *arXiv preprint arXiv:2412.06559*.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, Zheyan Luo, Zhangchi Feng, and Yongqiang Ma. 2024b. Llamafactory: Unified efficient fine-tuning of 100+ language models. *arXiv preprint arXiv:2403.13372*.

Wanjun Zhong, Ruixiang Cui, Yiduo Guo, Yaobo Liang, Shuai Lu, Yanlin Wang, Amin Saied, Weizhu Chen, and Nan Duan. 2023. Agieval: A human-centric benchmark for evaluating foundation models. *arXiv preprint arXiv:2304.06364*.

A Additional Implementation Details

A.1 Training

We train all models using LLaMA-Factory (Zheng et al., 2024b) on 8 NVIDIA A100-SXM4-40G GPUs. We use full parameter fine-tuning across all the models in our experiment. Training hyperparameters are detailed in Table 4.

Hyperparameter	Value
Precision	BF16
Optimization	Flash Attention2
Max Token Length	8192
Batch Size	128
Learning Rate	5×10^{-6}
LR Scheduler	Linear
Warmup Ratio	0.03
Weight Decay	0.0
Epochs	1.0
DeepSpeed	ZeRO-3

Table 4: Training hyperparameters used for the experiments.

A.2 Code Execution Filtering

Before code execution, we filter out data instances where the solution code uses randomization libraries or the input is excessively large, to ensure deterministic and stable execution.

During code execution, to reduce execution time and computational overhead when executing codes at scale, we discard any data instance where code execution exceeds 5 seconds or results in a runtime error. Also, to avoid excessively long execution traces, we filter out execution traces with more than 300 lines.

A.3 Data Generation Configuration

We use the default generation configuration of the `Qwen3-32B` translator. Specifically, we set the maximum token length to 16,382, with a temperature of 0.6, a top-p value of 0.95, and a top-k value of 20.

A.4 Evaluation Configuration

During evaluation, we set the temperature to 0.0 and use a maximum token length of 16,382. We enable the `enable_thinking=True` option to allow the model to think before generating solutions.

A.5 Prompt Templates

We present the prompt templates used in the experiment in Table 8.

B Execution Trace Example

Table 7 presents an example of executable code alongside its execution trace generated by `Snoop`. To enable tracing, the `@snoop` decorator must be applied to the main entry function. The resulting

execution trace includes function calls and return values, executed lines annotated with line numbers, and updated variable values.

C Additional Experiment Results

We present the experiment results for `Qwen3-8B` in Table 6. Similar to the results of `Qwen3-4B`, our method overall outperforms all the baselines with particular strength in the coding benchmark.

Additionally, we present token length analysis of `Qwen3-8B` on LiveBench in Table 5

Method	Avg Token	Max Token Reached
No Training	9030	116
CodeI/O	7362	83
Ours	6289	54

Table 5: Token length statistics on the LiveBench evaluation using `Qwen3-8B`.

D Licenses

We include the licenses of the datasets and models we used in this work.

Dataset License:

- LiveBench: Apache-2.0

Model Licenses:

- `Qwen3-4B`: Apache-2.0
  - https://huggingface.co/Qwen/Qwen3-4B
- `Qwen3-8B`: Apache-2.0
  - https://huggingface.co/Qwen/Qwen3-8B
- `Qwen3-32B`: Apache-2.0
  - https://huggingface.co/Qwen/Qwen3-32B

Methods	LiveBench			MATH500	BBH	AGIEval	GPQA	Avg
	Code	Math	Reasoning					
No Training	45.8	**64.4**	67.1	88.0	74.4	31.5	36.3	58.2
Code	37.5	14.4	50.4	86.6	63.9	**32.1**	35.2	45.7
Raw Trace	33.5	20.6	22.8	86.0	64.5	31.2	33.7	41.7
CodeI/O	37.1	62.2	**69.8**	87.6	77.0	32.0	39.5	57.8
Ours	**58.2**	63.1	69.2	**88.8**	**78.6**	31.9	**40.8**	**61.5**

Table 6: Experiment results (accuracy) of `Qwen3-8B`. Bolded scores indicate the highest performance.

Executable Code

```
import snoop # Import Snoop library

@snoop # Add the decorator to trace the function
def main_solution(num):
    if num < 0:
        return '-' + str(main_solution(-num))
    elif num < 7:
        return str(num)
    else:
        return str(main_solution(num // 7)) + str(
            num % 7)
main_solution(num=100) # Function call
```

Execution Trace generated by Snoop

```
>>> Call to main_solution # Function call
 ...... num = 100 # Input
   38 | def main_solution(num):
   39 |     if num < 0: # Executed line of code
   41 |     elif num < 7:
   44 |         return str(main_solution(num // 7))
        + str(num % 7)
     >>> Call to main_solution
     ...... num = 14 # Variable value
       38 | def main_solution(num):
       39 |     if num < 0:
       41 |     elif num < 7:
       44 |         return str(main_solution(num //
            7)) + str(num % 7)
         >>> Call to main_solution
         ...... num = 2
           38 | def main_solution(num):
           39 |     if num < 0:
           41 |     elif num < 7:
           42 |         return str(num)
         <<< Return value from main_solution: '2'
       44 |         return str(main_solution(num //
            7)) + str(num % 7)
     <<< Return value from main_solution: '20'
   44 |         return str(main_solution(num // 7))
        + str(num % 7)
 <<< Return value from main_solution: '202' # Return
```

Table 7: An example of an execution trace generated by the Python tool called `Snoop`

Execution Trace Translation Template

Given a question, an input to the question, and an execution trace that solves the question, your job is to translate the execution trace into a step-by-step thinking process. Here are some rules for translation:
- Use the exact values from the execution trace during the thought process to ensure the correctness of the thought process.
- Do not write code in your thinking process.
- Pretend you are not given the execution trace and you are solving the question by tracing the code by yourself. So, you should not mention that you are following the execution trace even when you are thinking.

Question
{question}
Input
{input}
Execution Trace
‘‘‘
{trace}
‘‘‘

CodeI/O Solution Generation Template

You are given a question that requires some input and output variables as follows:
{question}
—-
You are also given a solution code that solves the question:
{code}
—-
Given the following input:
{input}
Predict the output of the question by tracing the given solution code step by step to reach the final output.

User Prompt Template for Raw Execution Trace Baseline

You are given a question that requires some input and output variables as follows:
{question}
—-
Here is the solution code that solves the question:
‘‘‘ {code} ‘‘‘
Given the following input:
{input}
Generate a step-by-step execution trace of by tracing the given solution code step by step to reach the final output.

User Prompt Template for Code Generation Baseline

You are given a question that requires some input and output variables as follows:
{question}
—-
Generate a solution code that solves the question.

User Prompt Template for CodeI/O and Our Method

You are given a question that requires some input and output variables as follows:
{question}
—-
You are also given a solution code that solves the question:
{code}
—-
Given the following input:
{input}
Predict the output of the question by tracing the given solution code step by step to reach the final output.

Table 8: Prompt Templates used in the experiments

Problem: Given a string consisting of lowercase letters, what are all the possible unique permutations of the string?

Our Solution	CodeI/O Solution
To solve the problem of generating all unique permutations of the input string `'hrf'`, we follow these steps: 1. **Convert the string to a list of characters**: The input string `'hrf'` is split into individual characters: `['h', 'r', 'f']`. This allows us to rearrange the characters systematically. 2. **Generate all permutations**: Using a permutation algorithm or library function, we compute all possible orderings of the characters. For a 3-character string with unique elements, there are $3! = 6$ permutations. The permutations are: - ('h', 'r', 'f') - ('h', 'f', 'r') - ('r', 'h', 'f') - ('r', 'f', 'h') - ('f', 'h', 'r') - ('f', 'r', 'h') 3. **Convert permutations to strings**: Each permutation tuple is joined into a string: - ('h', 'r', 'f') -> 'hrf' - ('h', 'f', 'r') -> 'hfr' - ('r', 'h', 'f') -> 'rhf' - ('r', 'f', 'h') -> 'rfh' - ('f', 'h', 'r') -> 'fhr' - ('f', 'r', 'h') -> 'frh' 4. **Return the final list**: The resulting list of unique permutations is: `['hrf', 'hfr', 'rhf', 'rfh', 'fhr', 'frh']`. This approach ensures all permutations are generated efficiently, leveraging the uniqueness of the input characters to avoid duplicates.	The input string is 'hrf', which consists of three distinct lowercase letters. The solution code generates all possible permutations of these characters. Here's the step-by-step breakdown: 1. **Convert the input string to a list of characters**: char_list = ['h', 'r', 'f'] 2. **Generate permutations**: Using itertools.permutations, the code calculates all possible orderings of the three unique characters. Since there are no duplicates, the total permutations are $3! = 6$: - ('h', 'r', 'f') - ('h', 'f', 'r') - ('r', 'h', 'f') - ('r', 'f', 'h') - ('f', 'h', 'r') - ('f', 'r', 'h') 3. **Convert tuples to strings**: Each permutation tuple is joined into a string: - 'hrf' - 'hfr' - 'rhn' » ERROR: 'rhf' - 'rfh' - 'fhr' - 'frh' 4. **Final output**: The list of unique permutations is returned as: python ['hrf', 'hfr', 'rhf', 'rfh', 'fhr', 'frh'] **Answer:** The output is a list containing all 6 unique permutations of the input string 'hrf', as shown above.

Table 9: An example of CodeI/O where the intermediate step is incorrect but the final output is correct. Our solution refers to the exact permutation values in the execution trace, which makes it easier to get the intermediate steps correct.

Problem: In a historical game of elimination, a group of people stand in a circle. Starting from a given person, every k-th person is eliminated until only one person remains. Given the number of people in the circle and the step count, which position in the circle will be the last remaining person? The input is {'n': 17, 'k': 3}

Our Solution

```
To solve the Josephus problem for `n = 17` and `k =
    3`, we simulate the elimination process step by
     step. Here's the reasoning process:

1. **Initial Setup**:
   The people are arranged in a circle with
       positions `[1, 2, 3, ..., 17]`. The
       elimination starts at position 1, and every
       3rd person is removed.

2. **Recursive Elimination**:
   The Josephus problem uses the formula:
   $$
   J(n, k) = (J(n-1, k) + k) \mod n
   $$
   where $ J(1, k) = 0 $ (0-based index). Since the
       problem uses 1-based indexing, the final
       result is $ J(n, k) + 1 $.

3. **Step-by-Step Calculation**:
   Compute $ J(n, 3) $ for $ n = 1 $ to $ 17 $:
   - $ J(1) = 0 $
   - $ J(2) = (0 + 3) \mod 2 = 1 $
   - $ J(3) = (1 + 3) \mod 3 = 1 $
   - $ J(4) = (1 + 3) \mod 4 = 0 $
   - $ J(5) = (0 + 3) \mod 5 = 3 $
   - $ J(6) = (3 + 3) \mod 6 = 0 $
   - $ J(7) = (0 + 3) \mod 7 = 3 $
   - $ J(8) = (3 + 3) \mod 8 = 6 $
   - $ J(9) = (6 + 3) \mod 9 = 0 $
   - $ J(10) = (0 + 3) \mod 10 = 3 $
   - $ J(11) = (3 + 3) \mod 11 = 6 $
   - $ J(12) = (6 + 3) \mod 12 = 9 $
   - $ J(13) = (9 + 3) \mod 13 = 12 $
   - $ J(14) = (12 + 3) \mod 14 = 1 $
   - $ J(15) = (1 + 3) \mod 15 = 4 $
   - $ J(16) = (4 + 3) \mod 16 = 7 $
   - $ J(17) = (7 + 3) \mod 17 = 10 $.

   Adjusting to 1-based indexing: $ 10 + 1 = 11 $.

4. **Final Result**:
   After all eliminations, the last remaining person
        is at position **11**.

**Answer**:
The last remaining person is in position `11`.
```

CodeI/O Solution

```
To solve the Josephus problem for `n = 17` and `k =
    3`, we simulate the recursive elimination
    process as implemented in the provided code.
    The goal is to determine the position of the
    last remaining person in a circle of 17
    individuals, where every 3rd person is
    eliminated in a clockwise fashion.

---

### **Understanding the Code Logic**

The recursive function `josephus_problem(array, k,
    index)` works as follows:

- **Base Case:** When only one person remains (`len(
    array) == 1`), return that person's position.
- **Recursive Step:**

  - Calculate the index of the next person to be
      eliminated using `(index + k) % len(array)`.
  - Remove the person at that index.
  - Call the function recursively with the updated
      array and the new index (which is the index
      of the next person after the eliminated one).

In the `main_solution(n, k)` function:
- An array is initialized with positions from `1` to
     `n`.
- The recursive function is called with `k - 1` to
    adjust for zero-based indexing.

---

### **Step-by-Step Simulation**

We simulate the recursive calls manually for `n =
    17`, `k = 3`, and track the eliminations:

1. Start with array: `[1, 2, 3, 4, 5, 6, 7, 8, 9,
    10, 11, 12, 13, 14, 15, 16, 17]`, index = 0, k
    = 2.
2. Eliminate index `(0 + 2) % 17 = 2` -> remove `3`.
3. Eliminate index `(2 + 2) % 16 = 4` -> remove `6`.
...
16. Eliminate index `(3 + 2) % 3 = 0` -> remove `4`.
17. Eliminate index `(0 + 2) % 2 = 0` -> remove
    `11`.

At this point, only `13` remains.

---

### **Final Answer**
The last remaining person is at position **13**.
```

Table 10: An example of CodeI/O where an intermediate reasoning is wrong and the final answer is incorrect. In contrast, our solution correctly derives the formula, as shown in the execution trace, whereas CodeI/O uses an incorrect one.