

代码执行作为 LLM 推理的基础监督

Dongwon Jung¹ Wenxuan Zhou² Muhao Chen¹

¹University of California, Davis, ²University of Southern California,
{ dwojung,muhchen }@ucdavis.edu zhouwenx@usc.edu

Abstract

使用链式思维 (CoT) 监督训练大型语言模型 (LLMs) 已被证明可以有效增强其推理能力。然而, 获得可靠和准确的推理监督仍然是一个重要的挑战。我们提出了一种可扩展方法, 通过利用程序执行的确定性来生成高质量的 CoT 监督数据集。与现有依赖昂贵人工标注或易出错的 LLM 生成 CoT 的推理数据集生成方法不同, 我们的方法从代码执行中提取可验证的逐步推理过程, 并将其转化为自然语言 CoT 推理。在各个领域的推理基准上的实验表明, 我们的方法能够有效地使 LLMs 在不同任务中具备可迁移的推理能力。此外, 消融研究验证了我们的方法能够生成高度准确的推理数据, 并通过减少无意义的重复和过度思考来降低推理时的总标记长度。¹

1 引言

大型语言模型 (LLMs) 在各种复杂的推理任务中展现了强大的性能。该领域的一个关键发展是链式思维 (CoT) 训练, 它通过鼓励在生成最终答案 (Chung et al., 2024; Ho et al., 2022; Magister et al., 2022; Li et al., 2023) 之前生成中间推理步骤来增强 LLMs。CoT 监督在提高 LLMs 的泛化能力和可解释性方面已被证明特别有效, 并已成为推理模型开发中的核心组成部分 (Ye et al., 2025; Muennighoff et al., 2025; Team, 2025; Chang et al., 2025)。

尽管取得了成功, 但大规模获取高质量的 CoT 数据仍然是监督推理大模型的一大挑战。现有的 CoT 数据集通常通过两种方式构建。首先, 人工标注的 CoT 示例 (Chung et al., 2024; Cobbe et al., 2021) 提供高质量且准确的推理指导, 但获取成本高且不可扩展。其次, 许多近期的努力依赖通过提示现有的大模型 (Magister et al., 2022; Muennighoff et al., 2025) 生成的自举 CoT 数据。然而, 这些合成数据通常存在中间推理错误、不一致性和缺乏基础性 (Zheng

et al., 2024a; Lyu et al., 2023; Chen et al., 2025)。尽管这些方法在过程或结果层面对 CoT 数据进行验证和过滤 (Zelikman et al., 2022; Lightman et al., 2023; Luo et al., 2024; Li et al., 2025), 但仍不足以保证中间推理步骤的正确性, 从而削弱监督信号的可靠性。

在这项工作中, 我们提出了一种可扩展的方法, 用于生成可验证的 CoT 数据, 以利用程序执行的确定性监督 LLM 的推理过程。我们的核心观点是, 当问题可以被形式化并用可执行代码解决时, 所得的执行轨迹 (Åkerblom et al., 2014) 提供了与任务一致的固有正确的逐步推理。这些轨迹提供了一种可验证且无错误的替代方案, 可替代 LLM 生成的 CoTs, 并可以作为可信的监督来源。

具体来说, 我们首先寻找开源 Python 程序, 并使用调试器执行它们, 以提取丰富的执行轨迹, 包括中间变量值、行级执行顺序和程序控制流。由于生成的原始执行轨迹缺乏自然语言推理结构, 我们采用 LLMs 将原始执行轨迹翻译成流畅的人类可读理由, 类似于自然的 CoT 数据, 有效地结合了执行的正确性保障与 LLMs 的表达能力。我们的方法提供了一种可扩展的、无标注的流水线, 用于生成高质量且准确的推理监督。

实验表明, 使用我们方法训练的 LLM 展示了其有效性, 与基线方法相比, 在编码、数学和推理任务中实现了稳健的表现。消融研究进一步确认了我们的方法通过减轻无意义的重复和过度思考, 提高了数据质量并减少了整体标记长度。

2 方法

2.1 问题设置

我们的目标是通过用准确且可验证的 CoT 推理轨迹监督来增强 LLM 的推理能力。形式上, 给定一个输入序列 $\mathbf{x} = [x_1, \dots, x_m]$, 一个 LLM p_{θ} 通过一系列中间的推理步骤 $\mathbf{s} = [s_1, \dots, s_l]$ 生成一个输出序列 $\mathbf{y} = [y_1, \dots, y_n]$

¹代码和数据可在 <https://github.com/luka-group/Execution-Grounded-Reasoning> 获取

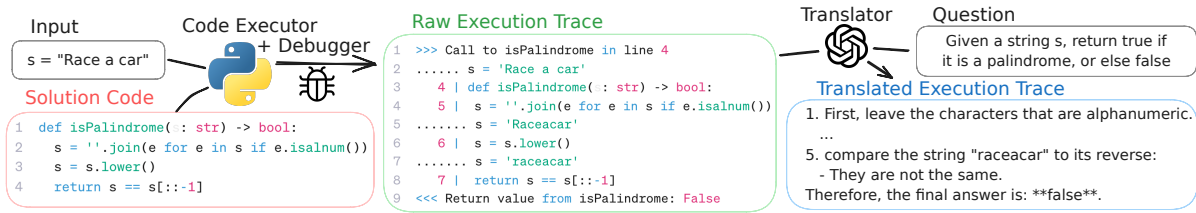


Figure 1: 我们方法的概述。经过转换的执行踪迹以代码执行为基础，使其成为对 LLM 进行推理监督的可靠且准确的来源。

。整体生成过程被定义为：

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

，其中模型首先在输入 \mathbf{x} 和之前步骤 $s_{<t}$ 的条件下生成每一个推理步骤 s ，接着基于完整的推理轨迹 \mathbf{s} 和原始输入 \mathbf{x} 生成最终答案 \mathbf{y} 。

高质量的 CoT 数据对于实现 LLMs (Lightman et al., 2023) 中强大的推理性能至关重要。为了大规模收集 CoT 数据，现有方法采用生成-过滤范式：首先使用 LLMs 采样 CoTs，然后过滤掉低质量的。结果级别的过滤通常检查最终答案 \mathbf{y} 是否与真实答案 (Xiong et al., 2025) 匹配，但这可能会漏掉那些凑巧产生正确答案的错误推理。过程级别的过滤更为理想，因为它评估中间推理质量，但对于当前的 LLMs 来说仍然具有挑战性 (Zheng et al., 2024a)。

在这项工作中，我们提出了一种根本不同的方法：从代码执行轨迹中构建 CoT 数据，这些数据本质上是逐步的、精确的，并与最终结果有因果关联，使它们成为准确和可验证推理监督的自然来源。在接下来的部分中，我们描述了如何构建高质量的 CoT 数据，然后使用监督微调对 LLM 进行微调。

2.2 执行跟踪获取

为了高效获取可靠且准确的 CoTs，我们利用丰富的编码数据，这些数据以问题-解决方案代码对的形式提供监督。这些对允许我们在可执行程序中奠定推理监督，这些程序反映了正确的问题解决逻辑。具体来说，给定一个解决方案代码片段 c 和一个输入 $\mathbf{x} = [\mathbf{x}_q; \mathbf{x}_i]$ ，其中 \mathbf{x}_q 是自然语言问题描述， \mathbf{x}_i 是具体测试输入（例如，特定字符串或数值输入），我们使用执行跟踪工具执行 c 以获取返回的答案 \mathbf{y} 和详细的执行跟踪 $\mathbf{s}_{\text{trace}}$ ：

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

我们使用了一种名为 Snoop 的 Python 调试工具 (Hall, 2024) 作为执行跟踪工具，它记录了详细的逐行执行信号，包括函数调用和返回、

执行的代码行以及更新的局部变量值。Snoop 生成的执行跟踪示例在 Figure 1 中提供。² 结果数据集记为 $D_{\text{trace}} = (\mathbf{x}, \mathbf{s}_{\text{trace}}, \mathbf{y})$ ，其中包含可验证的准确执行跟踪 $\mathbf{s}_{\text{trace}}$ 以及每个实例的正确最终输出 \mathbf{y} ，这些都奠定于一个可验证的代码执行器。

2.3 执行追踪转换

尽管执行轨迹 $\mathbf{s}_{\text{trace}}$ 捕捉到了可验证的、逐步的问题解决逻辑，但其格式与自然语言的 CoT 推理有很大差异。因此，直接在这样的轨迹上进行微调可能会妨碍泛化，并有可能导致对其他推理任务的灾难性遗忘。为了更好地将监督信号与自然语言推理对齐，我们使用 Translator 将原始轨迹 $\mathbf{s}_{\text{trace}}$ 转换为自然语言的 CoT $\mathbf{s}_{\text{nl_trace}} = (s_1, \dots, s_m)$ ：

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

翻译者被提示通过心理追踪代码来模拟人类解决问题。要求其用自然语言表达每一步的推理，同时如实反映代码执行过程中观察到的精确值和逻辑。这确保了输出能够反映程序行为背后的精确推理——以执行为基础，但以自然的、直观的、逐步的思维方式来表述。生成的数据集 $D_{\text{nl_trace}} = (\mathbf{x}, \mathbf{s}_{\text{nl_trace}}, \mathbf{y})$ 提供了既可验证又在语言上与典型 LLM 数据一致的高质量 CoT 轨迹，使其非常适合进行监督微调。

3 实验

为了评估从代码执行轨迹获得的监督是否真正增强了推理能力，我们通过将我们的方法与专为提升 LLM 推理能力而设计的基准数据集进行比较，开展了全面的实验。在本节中，我们展示了实验细节并讨论了结果。

3.1 实验设置

我们选择了 PyEdu-R，这是从 Python-Edu 中选择的一部分数据，作为监督数据集的来源。PyEdu-R 专注于 STEM 相关问题，比如逻辑谜题、数学相关任务、科学计算以及系统建模。

²一个更具说明性的执行踪迹示例在 ?? 中提供

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: Qwen3-4B 的实验结果 (准确率)。加粗的分数表示最高性能。

由于原始数据仅包含代码，我们使用了预处理版本，该版本包含由 LLM 生成的问题以及输入输出对，已由 Li et al. (2025) 公开提供。我们通过在输入上运行代码获得执行轨迹，然后使用 Qwen3-32B 将这些轨迹翻译为 Translator。此过程产生了大约 15K 条数据实例。

基线 我们将我们的方法与几种基线进行比较，这些基线旨在通过微调增强 LLM 推理。训练设置保持不变，唯一的区别在于从源数据进行的数据集策划过程。比较的基线包括：(1) 无训练，其中评估基础 LLM 而不进行进一步的微调；(2) 代码生成，其中模型被训练生成解决方案代码 c ，给定问题 x_q ；(3) 原始执行跟踪，其中模型学习直接从代码 c 和输入 x_i 生成原始执行轨迹 s_{trace} ，而绕过自然语言翻译；以及 (4) 代码 I/O (Li et al., 2025)，其中模型在由教师模型生成的 CoT 轨迹 s_{teacher} 上进行训练，随后是二元输出正确性反馈。为保持一致性，我们使用在我们 Translator 中使用的相同模型作为此基线中的教师模型。

模型 我们使用两种目标模型：Qwen3-4B 和 Qwen3-8B，对我们的数据进行 SFT。我们使用 Qwen3-32B 作为我们方法中的翻译器和 CodeI/O 基准方法的教师模型。对于这两种方法，我们启用 `enable_thinking=True` 选项，并在思考阶段后提取输出，以获得翻译和 CoT 生成结果。

评估基准 我们在广泛采用的推理基准上评估我们的方法和基线，包括 MATH500 (Lewkowycz et al., 2022)、BBH (Suzgun et al., 2022)、AGIEval (Zhong et al., 2023) 和 GPQA (Rein et al., 2024)。此外，我们利用了 LiveBench (White et al., 2025)，这是一个最近发布的综合基准，包含多样化的任务类别。我们专注于数学、编码和推理类别，因为我们的主要目标是评估这些方法的推理能力。具体而言，我们使用 2024-11-25 发布的版本，这是目前最新的版本。

实验结果 Qwen3-4B 是在 Table 1 中呈现的。总体而言，我们的方法优于所有基线，证明了

其有效性。代码生成和原始执行轨迹的基线在各方面表现不佳。尽管两者都来自于基于代码的监督，但它们在自然语言推理格式上出现对齐问题。尤其是，仅基于原始轨迹或代码生成进行训练并不能使模型具备用自然语言进行逐步推理的能力，导致迁移受到限制，性能下降——特别是在要求大量推理的任务中。

CodeI/O 基线在数学推理方面仍然表现强劲，但在编码和科学领域表现明显下降。这可能是因为教师模型的推理结构主要针对数学进行了优化，并有效地被蒸馏到学生模型中，这可能是以牺牲其他领域的性能为代价的。相比之下，我们的方法在所有推理领域都取得了平衡的改进。

在本节中，我们展示了一些消融研究，以评估我们数据的质量以及其在减少推理过程中重复和过度思考方面的有效性。

为了评估我们方法生成的数据质量，我们评价 (1) 最终输出的正确性和 (2) 中间推理步骤的正确性。我们为 Qwen3-32B 提供生成的解决方案和真实答案，以确定其是否得到了正确的答案。为了评估中间推理步骤，我们随机选择 200 个样本，并使用强大的推理模型 OpenAI-o3 来识别推理过程中的任何错误。

Table 2 显示了我们的方法和 CodeI/O 在最终输出和中间推理步骤上的准确性。我们的方法在这两个指标上都比 CodeI/O 达到更高的准确性，特别是在中间步骤的准确性上有较大的优势。这表明，我们的方法产生了更准确的推理步骤，从而得出正确的最终输出，因为它是基于可靠的代码执行。

为了评估使用我们方法训练的模型的生成效率，我们在 Table 3 中比较了标记长度。我们展示了与无训练基线相比，使用我们方法训练的模型在 Qwen3-4B 中生成的标记数减少了约 20%，而在 Qwen3-8B 中生成的标记数减少了约 30%。³ 此外，我们的方法显著减少了由于过度思考或重复输出导致模型达到最大标记限制的情况。重要的是，这种标记长度的减少不会影响性能，如在 ?? 所示。

³我们在 Table 5 中包含 Qwen3-8B 结果

Method	Output	Intermediate
CodeI/O	87.3	73.0
Ours	98.3	91.5

Table 2: 在 LiveBench 上比较最终输出的正确性准确率和中间推理步骤的正确性准确率。

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

Table 3: 使用 Qwen3-4B 在 LiveBench 评估上的符号长度统计。

为了进一步检查我们的数据，我们展示了在 Table 9 和 Table 10 中的两个推理追踪示例。第一个示例展示了一个案例，其中 CodeI/O 解决方案尽管在中间步骤出现了错误，仍然产生了正确的最终输出——具体来说，它在步骤 2 中错误地列出了“hrf”的排列——而我们的解决方案正确地完成了所有中间步骤。第二个示例显示了一个案例，其中 CodeI/O 解决方案的中间推理和最终输出都是错误的，因为在开始时使用了不正确的公式。相比之下，我们的解决方案在整个过程中产生了正确的计算，受到执行追踪的指导。

4 相关工作

Reasoning Distillation from Teacher Models 一种常见的方法是通过来自更强教师模型的推理链进行监督微调来提炼推理能力 (Ho et al., 2022; Magister et al., 2022; Li et al., 2023)。最近的工作利用测试时扩展将指令调优模型转变为元思维链 (Meta-CoT; Xiang et al. 2025)，即在解决问题之前首先生成思维令牌。我们的工作与利用测试时扩展的方法正交，旨在提高模型固有的逐步推理能力。

Training on Code for Reasoning 早期研究表明，在代码上训练的大型语言模型在各种推理任务中表现出色，包括常识推理 (Madaan et al., 2022)、因果推理 (Liu et al., 2023, 2024) 和数学推理 (Azerbaiyev et al., 2023; Shao et al., 2024)。然而，这些发现仅限于在代码上进行大量预训练的模型，且并未深入研究代码语义和结构如何影响推理能力。最近的研究利用代码执行轨迹来增强在编码任务上的推理，着重于漏洞检测、程序修复和代码生成等应用 (Ding et al., 2024b; Ni et al., 2024; Ding et al., 2024a)。相比之下，我们的工作针对的是超越代码领域的与任务无关的逐步推理。

最密切相关的工作是由 Li et al. (2025) 进行的，他们使用 CoT 推理来训练输入-输出预测任务的模型。虽然这两种方法都利用 CoT 进行输出预测，但我们的方法使用执行轨迹作为有根据的监督，而他们的方法仅依赖于二元输出正确性。

我们引入了一种新颖的方法，该方法利用代码执行跟踪作为可验证且易于扩展的监督机制，以增强大型语言模型 (LLM) 的逐步推理能力。我们的方法结合了执行的正确性与自然的链式推理步骤，为 LLM 提供高质量的推理监督。在各种推理任务中的实验表明，我们的方法优于之前的蒸馏方法，提供了一条可靠的途径，通过有根据的、无标注的监督来改进 LLM 的推理能力。

5

限制 尽管我们的方法提供了有根据且可靠的推理监督，但它本质上仅限于可以通过可执行代码表达和解决的任务。然而，我们已经证明我们的推理数据能够很好地迁移到其他领域，如数学和逻辑推理。此外，由于翻译依赖于一个大型语言模型，它可能并不总是完美的。尽管如此，我们通过评估最终输出和中间推理步骤，已经展示了我们数据的质量。

6

伦理声明 本工作遵循 ACL 道德规范。我们认为所呈现的工作没有直接相关的潜在风险。

References

- Beatrice Åkerblom, Jonathan Stendahl, Mattias Tumlén, 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*.
- 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*.
- Yanguibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. 2024a. Sem-coder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems*, 37:60275–60308.
- Yanguibo Ding, Benjamin Steenhoeck, 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. Deepseek-math: 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, Jingteng 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, Zheyang 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 附加实现细节

A.1 训练

我们使用 8 个 NVIDIA A100-SXM4-40G GPU 在 LLaMA-Factory (Zheng et al., 2024b) 上训练所有模型。在我们的实验中，我们对所有模型进行全参数微调。训练超参数详见 Table 4。

在代码执行之前，我们过滤掉那些使用随机化库的解决方案代码或输入过大的数据实例，以确保确定性和稳定的执行。

在代码执行过程中，为了减少大规模执行代码时的执行时间和计算开销，我们会丢弃任何代码执行超过 5 秒或导致运行时错误的数

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: 实验中使用的训练超参数。

例。此外，为避免过长的执行轨迹，我们会过滤掉超过 300 行的执行轨迹。

我们使用 Qwen3-32B 翻译器的默认生成配置。具体来说，我们将最大标记长度设置为 16,382，温度为 0.6，top-p 值为 0.95，top-k 值为 20。

在评估过程中，我们将温度设为 0.0，并使用最大令牌长度为 16,382。我们启用 `enable_thinking=True` 选项以允许模型在生成解决方案之前进行思考。

我们在 Table 8 中展示了实验中使用的提示模板。

Table 7 呈现了由 Snoop 生成的可执行代码示例及其执行跟踪。为了启用跟踪，必须将 `@snoop` 装饰器应用于主要入口函数。生成的执行跟踪包括函数调用和返回值，已执行的代码行附有行号，以及更新的变量值。

B 附加实验结果

我们展示了 Qwen3-8B 在 Table 6 中的实验结果。与 Qwen3-4B 的结果相似，我们的方法整体上优于所有基线，尤其在编码基准测试中表现突出。

此外，我们展示了 Qwen3-8B 在 LiveBench 上的标记长度分析 Table 5

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

Table 5: 在使用 Qwen3-8B 的 LiveBench 评估上的标记长度统计。

我们包括了本文中使用的数据集和模型的许可证。

数据集许可证：

- LiveBench: Apache-2.0

模型许可：

- 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
Code/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: Qwen3-8B 的实验结果 (准确性)。加粗的分数表示最高性能。

Executable Code	Execution Trace generated by Snoop
<pre>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) main_solution(num=100) # Function call</pre>	<pre>>>> 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) >>> 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) >>> 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) <<< Return value from main_solution: '20' 44 return str(main_solution(num // 7)) + str(num) <<< Return value from main_solution: '202' # Return</pre>

Table 7: 由名为 Snoop 的 Python 工具生成的执行跟踪示例

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 }  
““
```

Code/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 Code/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: 实验中使用的提示模板

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

Our Solution	CodeI/O Solution
<p>To solve the problem of generating all unique permutations of the input string <code>`hrf`</code>, we follow these steps:</p> <ol style="list-style-type: none"> Convert the string to a list of characters: The input string <code>`hrf`</code> is split into individual characters: <code>`['h', 'r', 'f']`</code>. This allows us to rearrange the characters systematically. 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: <ul style="list-style-type: none"> <code>`('h', 'r', 'f')`</code> <code>`('h', 'f', 'r')`</code> <code>`('r', 'h', 'f')`</code> <code>`('r', 'f', 'h')`</code> <code>`('f', 'h', 'r')`</code> <code>`('f', 'r', 'h')`</code> Convert permutations to strings: Each permutation tuple is joined into a string: <ul style="list-style-type: none"> <code>`('h', 'r', 'f')`</code> -> <code>`hrf`</code> <code>`('h', 'f', 'r')`</code> -> <code>`hfr`</code> <code>`('r', 'h', 'f')`</code> -> <code>`rhf`</code> <code>`('r', 'f', 'h')`</code> -> <code>`rfh`</code> <code>`('f', 'h', 'r')`</code> -> <code>`fhr`</code> <code>`('f', 'r', 'h')`</code> -> <code>`frh`</code> Return the final list: The resulting list of unique permutations is: <code>`['hrf', 'hfr', 'rhf', 'rfh', 'fhr', 'frh']`</code>. <p>This approach ensures all permutations are generated efficiently, leveraging the uniqueness of the input characters to avoid duplicates.</p>	<p>The input string is <code>`hrf`</code>, which consists of three distinct lowercase letters. The solution code generates all possible permutations of these characters. Here's the step-by-step breakdown:</p> <ol style="list-style-type: none"> Convert the input string to a list of characters: <code>char_list = ['h', 'r', 'f']</code> Generate permutations: Using <code>itertools.permutations</code>, the code calculates all possible orderings of the three unique characters. Since there are no duplicates, the total permutations are $3! = 6$: <ul style="list-style-type: none"> <code>`('h', 'r', 'f')`</code> <code>`('h', 'f', 'r')`</code> <code>`('r', 'h', 'f')`</code> <code>`('r', 'f', 'h')`</code> <code>`('f', 'h', 'r')`</code> <code>`('f', 'r', 'h')`</code> Convert tuples to strings: Each permutation tuple is joined into a string: <ul style="list-style-type: none"> <code>`('h', 'r', 'f')`</code> -> <code>`hrf`</code> <code>`('h', 'f', 'r')`</code> -> <code>`hfr`</code> <code>`('r', 'h', 'f')`</code> -> <code>`rhf`</code> <code>`('r', 'f', 'h')`</code> -> <code>`rfh`</code> <code>`('f', 'h', 'r')`</code> -> <code>`fhr`</code> <code>`('f', 'r', 'h')`</code> -> <code>`frh`</code> Final output: The list of unique permutations is returned as: python <code>`['hrf', 'hfr', 'rhf', 'rfh', 'fhr', 'frh']`</code> <p>Answer: The output is a list containing all 6 unique permutations of the input string <code>`hrf`</code>, as shown above.</p>

Table 9: 这是一个 CodeI/O 的例子，其中中间步骤不正确但最终输出是正确的。我们的解决方案参考了执行痕迹中的确切排列值，这使得更容易得到正确的中间步骤。

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	CodeI/O Solution												
<p>To solve the Josephus problem for $n = 17$ and $k = 3$, we simulate the elimination process step by step. Here's the reasoning process:</p> <ol style="list-style-type: none"> Initial Setup: The people are arranged in a circle with positions $[1, 2, 3, \dots, 17]$. The elimination starts at position 1, and every 3rd person is removed. Recursive Elimination: The Josephus problem uses the formula: $J(n, k) = (J(n-1, k) + k) \pmod n$ where $J(1, k) = 0$ (0-based index). Since the problem uses 1-based indexing, the final result is $J(n, k) + 1$. Step-by-Step Calculation: <table border="0"> <tr> <td>Compute</td> <td>$J(n, 3)$</td> <td>for</td> <td>n</td> </tr> <tr> <td></td> <td></td> <td></td> <td>to</td> </tr> <tr> <td></td> <td></td> <td></td> <td>17</td> </tr> </table> <ul style="list-style-type: none"> - $J(1) = 0$ - $J(2) = (0 + 3) \pmod 2 = 1$ - $J(3) = (1 + 3) \pmod 3 = 1$ - $J(4) = (1 + 3) \pmod 4 = 0$ - $J(5) = (0 + 3) \pmod 5 = 3$ - $J(6) = (3 + 3) \pmod 6 = 0$ - $J(7) = (0 + 3) \pmod 7 = 3$ - $J(8) = (3 + 3) \pmod 8 = 6$ - $J(9) = (6 + 3) \pmod 9 = 0$ - $J(10) = (0 + 3) \pmod 10 = 3$ - $J(11) = (3 + 3) \pmod 11 = 6$ - $J(12) = (6 + 3) \pmod 12 = 9$ - $J(13) = (9 + 3) \pmod 13 = 12$ - $J(14) = (12 + 3) \pmod 14 = 1$ - $J(15) = (1 + 3) \pmod 15 = 4$ - $J(16) = (4 + 3) \pmod 16 = 7$ - $J(17) = (7 + 3) \pmod 17 = 10$ <p>Adjusting to 1-based indexing: $10 + 1 = 11$.</p> Final Result: After all eliminations, the last remaining person is at position 11. <p>Answer: The last remaining person is in position 11.</p>	Compute	$J(n, 3)$	for	n				to				17	<p>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.</p> <p>--- Understanding the Code Logic The recursive function <code>josephus_problem(array, k, index)</code> works as follows:</p> <ul style="list-style-type: none"> Base Case: When only one person remains (<code>len(array) == 1</code>), return that person's position. Recursive Step: <ul style="list-style-type: none"> Calculate the index of the next person to be eliminated using <code>(index + k)</code>. 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). <p>In the <code>main_solution(n, k)</code> function:</p> <ul style="list-style-type: none"> An array is initialized with positions from <code>1</code> to <code>n</code>. The recursive function is called with <code>k - 1</code> to adjust for zero-based indexing. <p>--- Step-by-Step Simulation We simulate the recursive calls manually for $n = 17$, $k = 3$, and track the eliminations:</p> <ol style="list-style-type: none"> Start with array: <code>[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]</code>, <code>index = 0</code>, <code>k = 2</code>. Eliminate index <code>(0 + 2)</code> Eliminate index <code>(2 + 2)</code> ... Eliminate index <code>(3 + 2)</code> Eliminate index <code>(0 + 2)</code> <p>At this point, only <code>13</code> remains.</p> <p>--- Final Answer The last remaining person is at position 13.</p>
Compute	$J(n, 3)$	for	n										
			to										
			17										

Table 10: 一个 CodeI/O 的例子，其中中间推理错误，最终答案也是不正确的。相比之下，我们的解决方案正确地推导出了公式，如执行轨迹所示，而 CodeI/O 使用了一个错误的公式。