ARM: Discovering Agentic Reasoning Modules for Mathematical Problem-Solving

Bohan Yao*
University of Washington
ServiceNow

Shiva Krishna Reddy Malay
ServiceNow

Vikas Yadav
ServiceNow
{vikas.yadav}@servicenow.com

Abstract

Large Language Model (LLM)-powered Multi-agent systems (MAS) have achieved state-of-the-art results on complex mathematical reasoning tasks. Recent works have proposed techniques to automate the design of these systems, but they often perform poorly and require computationally expensive re-discovery of architectures for other domains. A critical insight is that simple Chain of Thought (CoT) reasoning often performs competitively with these complex systems, suggesting that the fundamental reasoning unit warrants further investigation. To this end, we present a new paradigm for automatic MAS design that pivots the focus to optimizing CoT reasoning. We introduce the **A**gentic **R**easoning **M**odule (ARM), an agentic generalization of CoT where each granular reasoning step is executed by a specialized reasoning module discovered through an evolutionary tree search. The resulting ARM acts as a versatile reasoning building block for solving multi-step math problems. On challenging math benchmarks, including MATH500, AIME25, and HMMT25, our approach significantly outperforms both manually designed MASes and state-of-the-art automatic MAS design methods. Crucially, reasoning systems built with ARM exhibit strong generalization across different foundation models, maintaining high performance without requiring model-specific re-optimization.

1 Introduction

While Multi-Agent Systems (MAS) leveraging Large Language Models (LLMs) have achieved state-of-the-art results on complex mathematical reasoning benchmarks (Park et al., 2023; Qian et al., 2023; Hong et al., 2023), recent studies reveal a surprising trend: a well-prompted single-agent Chain-of-Thought (CoT)Wei et al. (2022) baseline often performs on par with, or even outperforms, these complex architectures on frontier models (Wang et al., 2024; Yao & Yadav, 2025). These advanced systems orchestrate multiple agents that adopt specialized roles—such as a "*problem decomposer*", "*symbolic calculator*", or "*proof verifier*"—to collaboratively solve problems (Wu et al., 2023), with a trend towards their automatic discovery (Zhang et al., 2025; Kim et al., 2024). The continued competitiveness of the foundational CoT method (Wei et al., 2022) suggests that the core reasoning unit—the individual deductive step—is of paramount importance for mathematical problem-solving. However, recent work in automated MAS design has centered on discovering optimal agent roles and interaction topologies, while the underlying deductive step, the core of the CoT baseline, has

*Work done during internship at ServiceNow

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

largely remained unchanged (Hu et al., 2025; Zhang et al., 2025). Our work pivots from this trend to fundamentally enhance the CoT paradigm.

We introduce the Agentic Reasoning Module (ARM), a novel approach that elevates each reasoning step from a simple textual continuation into a structured, agentic block discovered via reflection-guided evolutionary search (Fernando et al., 2024; Agrawal et al., 2025). This work presents ARM as an evolved version of CoT for complex mathematical reasoning, demonstrating state-of-the-art performance on challenging benchmarks including MATH500 Lightman et al. (2023), AIME25 and HMMT25. Furthermore, we show that ARM is a generalizable module for step-by-step reasoning that is effective in other analytical domains such as science and logical reasoning (Appendix D), and provide a rigorous justification with detailed ablations for our effective search strategy.

2 Methodology: Discovering the Agentic Reasoning Module

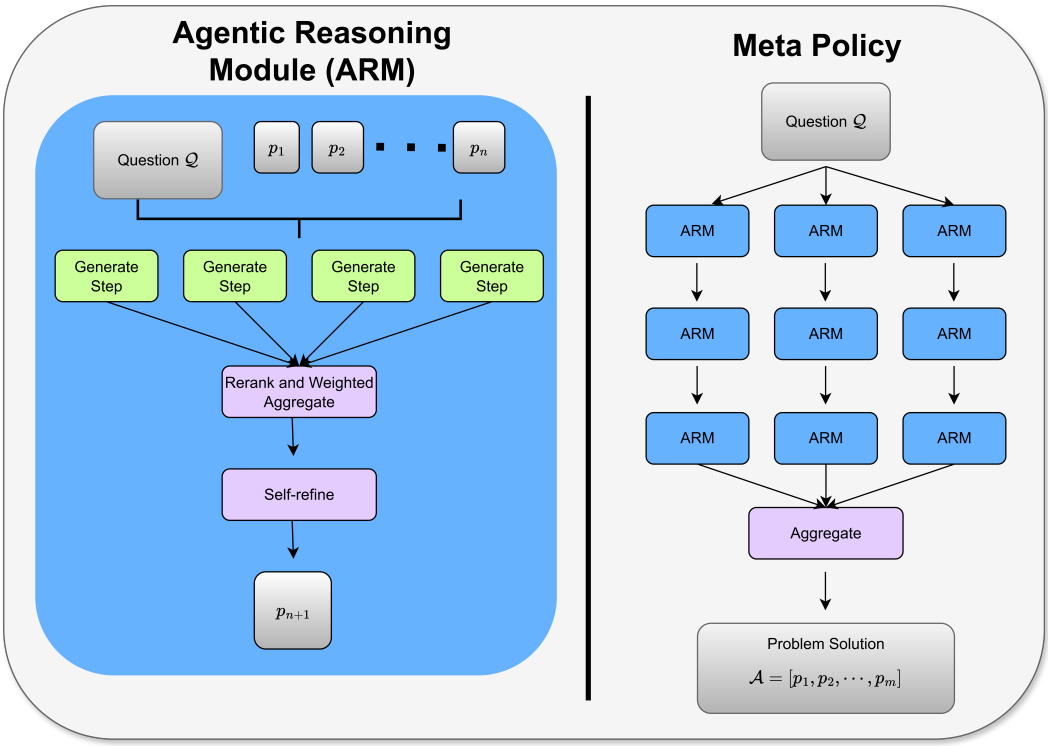

Figure 1: An illustration of the proposed ARM module on the left and the meta policy on the right using "Self refine" as an example MAS. The ARM module takes a question and previous reasoning steps and executes a MAS to get the next step. The meta policy uses ARM as a sub-module and orchestrates the overarching global strategy. Note that this is for illustration only, the actual step generator and the meta policy discovered by Algorithm-1 is more complex. See Appendix E for the Python implementation of the best discovered ARM by our algorithm. See Appendix F for the Python implementation of the best discovered meta-policy by our algorithm.

We introduce the **Agentic Reasoning Module (ARM)**, a structured, agentic replacement for a single step in a Chain of Thought (CoT) sequence (Wei et al., 2022). We model the problem-solving process by decomposing it into two key components: a **Step-Generator** ($m \in \mathcal{M}$) and a **Meta-Policy** ($\pi \in \Pi$). The step-generator, with signature $m : \mathcal{Q} \times \mathcal{P}^* \to \mathcal{P}$, takes a question q and a history of previous reasoning steps ($p_i \in \mathcal{P}$) to produce the next step. The meta-policy, $\pi : \mathcal{Q} \times \mathcal{M} \to \mathcal{A}$, is a higher-order program that orchestrates calls to a generator m to form a complete solution. Our work discovers ARM as a powerful, code-based implementation of the step-generator. Following prior work (Hu et al., 2025; Zhang et al., 2025), ARM is implemented as a self-contained Python class.

Within this framework, standard CoT can be viewed as a simple baseline pairing of a basic step generator (simple LLM call m_{CoT}) and a simple recursive policy (π_{Rec}) which iteratively calls m_{CoT} until completion. Our core motivation is to independently discover more powerful modules m^* (the ARM) and more powerful meta-policies π^* to significantly improve upon this baseline.

Discovering the Optimal Step-Generator (m^*): Our goal is to discover a superior step-generator, m^*, to replace the simple text generation step in m_{CoT}. Ideally, we would find m^* (ARM) by directly maximizing the expected reward $\mathcal{R}$ (final answer accuracy) over a full problem-solving trace generated by a recursive policy $\pi_{Rec}(q, m) = U^n_{m,q}(\emptyset)$, where U is the single step update operator $U_{m,q}(h) = h \cdot [m(q, h)]$ that appends the newly generated step to the reasoning history:

$$m^* = \underset{m \in \mathcal{M}}{\text{argmax}} \quad \mathbb{E}_{(q,a)\sim\mathcal{D}} \left[\mathcal{R} \left(\pi_{Rec} \left(q, m \right), a \right) \right]$$

However, this objective is intractable due to severe challenges in **credit assignment** over long sequence of steps and the **unconstrained search space** $\mathcal{M}$. To address this, we introduce a **scaffolded surrogate objective**. Instead of evaluating m on a full rollout generated by itself, we evaluate it within the stable context of a reference trace from the baseline m_{CoT}. We replace a small, contiguous block of l steps within this trace with our candidate module m, reformulating the problem as:

$$m^* = \underset{m \in \mathcal{M}}{\text{argmax}} \quad \mathbb{E}_{(q,a)\sim\mathcal{D}} \left[\mathcal{R} \left(U^*_{m_{CoT},q} \circ U^l_{m,q} \circ U^i_{m_{CoT},q}(\emptyset), a \right) \right]$$

where the starting index i is chosen randomly. This isolates the performance contribution of m to a small window, enabling direct credit assignment. Moreover, the surrounding CoT context provides a powerful inductive bias, constraining the search to modules that behave as effective, incremental reasoning step generators. We use $l = 3$, as it is sufficient to expose the module to critical compositional patterns (e.g., interactions with itself and the baseline m_{CoT}) while keeping the optimization computationally tractable.

Discovering the Optimal Meta-Policy (π^*): While an optimized step-generator m^* improves individual reasoning steps, the high-level meta-policy π that orchestrates them is equally critical. Searching for an optimal policy π^* by repeatedly evaluating candidates with the computationally expensive m^* is prohibitive. We therefore adopt a surrogate-based approach, searching for π^* using the fast, inexpensive baseline generator, m_{CoT}, as a proxy for m^*. This zero-shot transfer is effective because our discovery process is explicitly designed to produce an m^* that functions as a superior, "*drop-in*" replacement for m_{CoT}. A meta-policy that effectively orchestrates the simple steps of m_{CoT} is thus highly likely to generalize to the more powerful, but functionally analogous, steps of m^*. This enables the efficient discovery of sophisticated strategies, such as self-consistency or iterative refinement loops (Wang et al., 2023; Madaan et al., 2023), without incurring the high computational cost of executing the full m^* module.

Reflection-Guided Evolutionary Search: We discover both the optimal step-generator m^* and meta-policy π^* using a unified **Reflection-Guided Evolutionary Search**. This method performs a tree search over the programmatic space, beginning with a root node representing the baseline program (m_{CoT} or π_{Rec}). The search iteratively refines this tree through a three-step process: **1) Selection:** a promising parent program is chosen based on its validation performance. **2) Expansion:** a two-stage *Reviewer Agent* intelligently mutates the parent. A *Critic* first analyzes execution traces to identify logical errors or inefficiencies, then a *Designer* uses this critique to generate a new, syntactically valid program with a targeted modification. **3) Evaluation:** the new program's average reward, $\widetilde{\mathcal{R}}$, is computed using the appropriate surrogate objective (the scaffolded objective for a step-generator m, or a full rollout with m_{CoT} for a meta-policy π). This reflection-driven process ensures the search evolves programs purposefully, leading to a more efficient discovery of high-performance modules and policies (see Algorithm 1).

3 Experiments

Benchmarks: We evaluate our discovered modules on several challenging math benchmarks: MATH500 Lightman et al. (2023), *American Invitational Mathematics Examination* (*AIME25*[2])

[2] `https://huggingface.co/datasets/MathArena/aime_2025`

and the *February Harvard-MIT Mathematics Tournament* (*HMMT25*[3]) . Additionally, we found that ARM generalizes to other domains requiring rigorous, step-wise deduction such as science and logical reasoning. We report these results in Appendix D.

ARM: Our optimization process discovers the optimal step-generator module (m^*) and meta-policy (π^*) in two independent phases. We use a 1000-sample subset of the Open-R1-Mixture-of-Thoughts HuggingFace (2025) dataset for validation. To create this subset, we take the math and science splits of the original dataset, filter to samples where the provided Deepseek-R1 DeepSeek-AI et al. (2025) reasoning trace had a length of between 8k and 10k tokens (to filter to samples of appropriate difficulty), and randomly sample 1000 problems from the filtered problems.

First, we discover the ARM module (m^*) by employing our Reflection-Guided Evolutionary Search (Algorithm 1) to optimize the scaffolded surrogate objective. We then independently discover the meta-policy (π^*) using the same evolutionary search algorithm. For computational tractability, this second search uses the simple baseline module, m_{CoT}, as an efficient surrogate for the more complex m^*, as justified previously. We evaluate the discovered m^* and π^* with maximal validation accuracies across all benchmarks, without any task-specific re-optimization.

We run both the ARM module optimization and the meta-policy optimization for 20 iterations. Both optimizations are performed using GPT-4.1-nano as the MAS executor model. Whenever sampling from the MAS executor model, we use a temperature of 1.0 with a top_p of 0.95.

Baselines: We compare our methodology against two categories of multi-agent systems (MAS): prominent handcrafted systems and leading automated MAS generation approaches. Our handcrafted baselines include **Chain of Thought (CoT)** Wei et al. (2022), its ensemble variant **CoT-Self Consistency (CoT-SC)** (Wang et al., 2023), the iterative **Self-Refine**Madaan et al. (2023) method, and the multi-perspective **LLM-Debate** Du et al. (2023). For automated MAS generation, we benchmark against **ADAS** Hu et al. (2025) and **AFlow** Zhang et al. (2025), which search for optimal agent roles and interaction topologies. We evaluate the performance of ADAS and AFlow using both the original optimization configuration of using a 20% split of the test dataset as the validation dataset (resulting in a benchmark-optimized MAS for each benchmark) and using the ARM optimization configuration of using the 1000-sample subset of Open-R1-Mixture-of-Thoughts HuggingFace (2025) as the validation dataset (resulting in a single MAS which we evaluate across all benchmarks without benchmark-specific re-optimization). We denote baselines of the former configuration using "(test set)" and baselines of the latter configuration using "(1000-sample)" in the main results below. See Appendix G for more details on the baseline implementations.

Models: We use OpenAI's o4-mini-high OpenAI (2025b) reasoning model as the MAS designer for both the baselines ADAS, AFlow, and our method ARM, as MAS generation requires frontier performance in coding and instruction following. We test three models as backbone LLMs executing the MAS: two closed source models GPT-4.1-nano OpenAI (2025a), GPT-4o OpenAI et al. (2024) and one open source model Llama-3.3-70B Meta (2024).

4 Results

Method	GPT-4.1-nano				GPT-4o				LLaMA-3.3-70B			
	MATH500	AIME25	HMMT25	Average	MATH500	AIME25	HMMT25	Average	MATH500	AIME25	HMMT25	Average
CoT	82.0	15.1	9.9	35.7	75.0	7.3	0.5	27.6	75.0	6.8	3.1	28.3
CoT-SC	**86.2**	21.9	13.5	40.5	81.8	12.5	2.1	32.1	78.5	4.2	5.7	29.5
Self-Refine	84.2	17.2	9.4	36.9	77.2	6.8	2.6	28.9	77.8	6.8	4.2	29.6
LLM-Debate	84.2	15.1	16.7	38.7	81.8	9.9	3.1	31.6	79.0	5.7	4.2	29.6
ADAS (test set)	79.8	12.0	5.2	32.3	65.5	1.0	0.0	22.2	67.2	3.1	0.0	23.4
ADAS (1000-sample)	77.3	0.0	6.8	28.0	69.0	0.0	0.5	23.3	22.2	3.1	0.5	8.6
AFlow (test set)	74.5	18.8	12.0	35.1	75.5	9.9	3.6	29.7	65.2	4.7	0.0	23.3
AFlow (1000-sample)	77.0	16.7	10.4	34.7	48.8	9.4	0.0	19.4	63.2	7.2	3.1	24.5
ARM **(Ours)**	82.0	18.2	14.6	38.3	78.3	13.5	5.7	32.5	80.0	**8.3**	5.2	31.2
ARM + MP **(Ours)**	86.0	**23.4**	**22.4**	**43.9**	**82.0**	**17.2**	**9.4**	**36.2**	**80.8**	7.8	**6.8**	**31.8**

Table 1: Main results on three complex math reasoning benchmarks across three foundation models. We compare against two groups of baselines: (1) foundational reasoning strategies used to build agentic systems (CoT, CoT-SC, Self-Refine, and LLM-Debate), and (2) existing state-of-the-art automatic MAS design methods (ADAS and AFlow). Our approach is presented in two variants: **ARM**, which recursively applies the discovered reasoning module, and our full method, **ARM + MP**, which combines the ARM with a learned Meta-Policy (MP). Best score in each category is **bolded** and second best score is underlined.

[3] `https://huggingface.co/datasets/MathArena/hmmt_feb_2025`

We summarize our results in Table 1 and the key findings are as follows:

(1) Foundational Operators outperform MAS: Our results reveal a crucial insight: foundational reasoning methods like Chain-of-Thought (CoT) consistently outperform complex, automatically-generated multi-agent systems (MAS) such as AFlow and ADAS. This strongly suggests that the primary driver of performance is the quality of the granular, step-by-step reasoning, not just the complexity of the high-level architecture. A sophisticated orchestration layer cannot compensate for—and may even hinder—a flawed deductive process.

(2) ARM achieving top performance: ARM consistently outperforms all of the operator baselines especially in two harder datasets: AIME and HMMT. This can be attributed to our ARM-based approach revitalizing the powerful CoT paradigm by augmenting its core deductive step with a powerful agentic block. This demonstrates the effectiveness of enhancing the core deductive step rather than focusing solely on high-level architectural complexity. Consequently, ARM consistently achieves state-of-the-art performance, outperforming both handcrafted operators and automatic MAS baselines across all evaluated mathematical datasets.

5 Analyses

We performed two analyses to empirically validate our discovery process, with theoretical justifications in Appendix-B. Firstly, to confirm the scaffolded objective improves per-step competence (Appendix-B.4), a targeted ablation executed the top five discovered modules for a single step from critical junctures within m_{CoT} traces, as identified by an LLM-judge (`openai/gpt-oss-20b`). The results (Figure 3) show a module's rank strongly correlates with a lower per-step error rate, confirming our search discovers granularly robust modules. Secondly, we validate our decoupled training strategy by showing the meta-policy's zero-shot transfer from the m_{CoT} surrogate to the final ARM (m^*). To disentangle the two theoretical sources of gain (Appendix-B.5)—a superior module and its ability to find a better reasoning path—we compare the meta-policy with (1) the baseline m_{CoT}, (2) m^* taking over from baseline-generated states, and (3) the full system with m^*. The full system's superior performance (Figure 2) confirms gains from both factors, validating our decoupled search.

Meta Policy Name (abbreviated)	CoT Baseline	CoT→Meta	Meta Policy
VWASCCoT	35.1%	33.7%	42.0%
CWDCWACCCoT	37.2%	39.3%	41.8%
RVDCCWASCCoT	33.7%	40.0%	41.8%
DRWASCCoT	35.5%	34.9%	41.8%
MBECDCCWASCCoT	36.3%	39.2%	41.4%

Figure 2: Validation of the meta-policy transfer for top discovered policies. The 'CoT Baseline' column shows the performance of the discovered meta-policy when paired with the simple m_{CoT} surrogate module. The 'Meta Policy' column shows the performance of that same meta-policy when paired with the powerful ARM module m^*. The intermediate **CoT→Meta** column isolates the performance gain from the superior m^* module by evaluating it on states (progress) generated by 'CoT Baseline'.

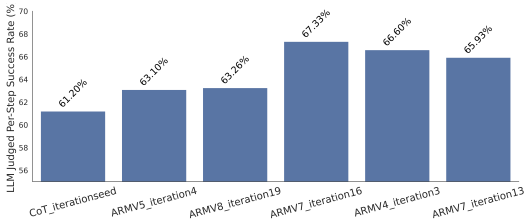

Figure 3: Comparison of LLM judged per-step success rates between the baseline *Chain-of-Thought* (CoT) and multiple *ARM* variants. CoT appears first, followed by ARM variants ordered by final performance.

6 Conclusion

We introduced ARM, a modular agentic reasoning framework that revitalizes the traditional Chain-of-Thought (CoT) paradigm by augmenting it with multi-agentic system modules. Through extensive experiments, we demonstrated that ARM consistently advances the performance of CoT on math reasoning tasks, especially when combined with a strong meta-policy orchestrating the ARM calls. Beyond empirical improvements, ARM sheds light on an important perspective: improving the granular step by step reasoning process holds the key to progress in this domain. By preserving the simplicity and generality of CoT steps, while enhancing its reasoning depth and modularity, ARM provides a versatile and powerful foundation that can be applied across tasks and models. ARM represents a step toward a robust and broadly applicable modular reasoning approach with LLMs, paving the way for future research to focus on discovering powerful, reusable reasoning units as a core component of agentic systems.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Fu-Chieh Chang, Yu-Ting Lee, Hui-Ying Shih, Yi Hsuan Tseng, and Pei-Yuan Wu. Rl-star: Theoretical analysis of reinforcement learning frameworks for self-taught reasoner, 2025. URL `https://arxiv.org/abs/2410.23912`.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL `https://arxiv.org/abs/2501.12948`.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate, 2023. URL `https://arxiv.org/abs/2305.14325`.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235, pp. 8370–8386. PMLR, 2024. URL `https://proceedings.mlr.press/v235/fernando24a.html`.

Sirui Hong, Xiawu Zheng, Jonathan Chen, K Yang, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. In *International Conference on Learning Representations (ICLR)*, 2025.

HuggingFace. Open r1: A fully open reproduction of deepseek-r1, January 2025. URL `https://github.com/huggingface/open-r1`.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

Sham Kakade and John Langford. Approximately optimal approximate reinforcement learning. In *Proceedings of the Nineteenth International Conference on Machine Learning*, ICML '02, pp. 267–274, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. ISBN 1558608737.

Juno Kim, Denny Wu, Jason Lee, and Taiji Suzuki. Metastable dynamics of chain-of-thought reasoning: Provable benefits of search, rl and distillation, 2025. URL https://arxiv.org/abs/2502.01694.

Sungwoo Kim, Lin Xu, Yifan Guo, Arif Rahman, and Shiyi Wang. Aflow: Automating agentic workflow generation for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2024. Accessed: YYYY-MM-DD.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Aman Madaan, Niket Tandon, Prakhar Gupta, Kevin Hall, Luyu Gao, Sarah Wiegreffe, Uri Alon, Pengcheng Cair, et al. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*, 2023.

Meta. Llama 3.3 model card. https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/, December 2024. Accessed: 2025-09-27.

OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, April 2025a. Accessed: 2025-09-27.

OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, April 2025b. Accessed: 2025-09-27.

OpenAI, :, Aaron Hurst, Adam Lerer, Adam P. Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, Aleksander Mądry, Alex Baker-Whitcomb, Alex Beutel, Alex Borzunov, Alex Carney, Alex Chow, Alex Kirillov, Alex Nichol, Alex Paino, Alex Renzin, Alex Tachard Passos, Alexander Kirillov, Alexi Christakis, Alexis Conneau, Ali Kamali, Allan Jabri, Allison Moyer, Allison Tam, Amadou Crookes, Amin Tootoochian, Amin Tootoonchian, Ananya Kumar, Andrea Vallone, Andrej Karpathy, Andrew Braunstein, Andrew Cann, Andrew Codispoti, Andrew Galu, Andrew Kondrich, Andrew Tulloch, Andrey Mishchenko, Angela Baek, Angela Jiang, Antoine Pelisse, Antonia Woodford, Anuj Gosalia, Arka Dhar, Ashley Pantuliano, Avi Nayak, Avital Oliver, Barret Zoph, Behrooz Ghorbani, Ben Leimberger, Ben Rossen, Ben Sokolowsky, Ben Wang, Benjamin Zweig, Beth Hoover, Blake Samic, Bob McGrew, Bobby Spero, Bogo Giertler, Bowen Cheng, Brad Lightcap, Brandon Walkin, Brendan Quinn, Brian Guarraci, Brian Hsu, Bright Kellogg, Brydon Eastman, Camillo Lugaresi, Carroll Wainwright, Cary Bassin, Cary Hudson, Casey Chu, Chad Nelson, Chak Li, Chan Jun Shern, Channing Conger, Charlotte Barette, Chelsea Voss, Chen Ding, Cheng Lu, Chong Zhang, Chris Beaumont, Chris Hallacy, Chris Koch, Christian Gibson, Christina Kim, Christine Choi, Christine McLeavey, Christopher Hesse, Claudia Fischer, Clemens Winter, Coley Czarnecki, Colin Jarvis, Colin Wei, Constantin Koumouzelis, Dane Sherburn, Daniel Kappler, Daniel Levin, Daniel Levy, David Carr, David Farhi, David Mely, David Robinson, David Sasaki, Denny Jin, Dev Valladares, Dimitris Tsipras, Doug Li, Duc Phong Nguyen, Duncan Findlay, Edede Oiwoh, Edmund Wong, Ehsan Asdar, Elizabeth Proehl, Elizabeth Yang, Eric Antonow, Eric Kramer, Eric Peterson, Eric Sigler, Eric Wallace, Eugene Brevdo, Evan Mays, Farzad Khorasani, Felipe Petroski Such, Filippo Raso, Francis Zhang, Fred von Lohmann, Freddie Sulit, Gabriel Goh, Gene Oden, Geoff Salmon, Giulio Starace, Greg Brockman, Hadi Salman, Haiming Bao, Haitang Hu, Hannah Wong, Haoyu Wang, Heather Schmidt, Heather Whitney, Heewoo Jun, Hendrik Kirchner, Henrique Ponde de Oliveira Pinto, Hongyu Ren, Huiwen Chang, Hyung Won Chung, Ian Kivlichan, Ian O'Connell, Ian O'Connell, Ian Osband, Ian Silber, Ian Sohl, Ibrahim Okuyucu, Ikai Lan, Ilya Kostrikov, Ilya Sutskever, Ingmar Kanitscheider, Ishaan Gulrajani, Jacob Coxon, Jacob Menick, Jakub Pachocki, James Aung, James Betker, James Crooks, James Lennon, Jamie Kiros, Jan Leike, Jane Park, Jason Kwon, Jason Phang, Jason Teplitz, Jason Wei, Jason Wolfe, Jay Chen, Jeff Harris, Jenia Varavva, Jessica Gan Lee, Jessica Shieh, Ji Lin, Jiahui Yu, Jiayi Weng, Jie Tang, Jieqi Yu, Joanne Jang, Joaquin Quinonero Candela, Joe Beutler, Joe Landers, Joel Parish, Johannes Heidecke, John Schulman, Jonathan Lachman, Jonathan McKay, Jonathan Uesato, Jonathan Ward, Jong Wook Kim, Joost Huizinga, Jordan Sitkin, Jos Kraaijeveld, Josh

Gross, Josh Kaplan, Josh Snyder, Joshua Achiam, Joy Jiao, Joyce Lee, Juntang Zhuang, Justyn Harriman, Kai Fricke, Kai Hayashi, Karan Singhal, Katy Shi, Kavin Karthik, Kayla Wood, Kendra Rimbach, Kenny Hsu, Kenny Nguyen, Keren Gu-Lemberg, Kevin Button, Kevin Liu, Kiel Howe, Krithika Muthukumar, Kyle Luther, Lama Ahmad, Larry Kai, Lauren Itow, Lauren Workman, Leher Pathak, Leo Chen, Li Jing, Lia Guy, Liam Fedus, Liang Zhou, Lien Mamitsuka, Lilian Weng, Lindsay McCallum, Lindsey Held, Long Ouyang, Louis Feuvrier, Lu Zhang, Lukas Kondraciuk, Lukasz Kaiser, Luke Hewitt, Luke Metz, Lyric Doshi, Mada Aflak, Maddie Simens, Madelaine Boyd, Madeleine Thompson, Marat Dukhan, Mark Chen, Mark Gray, Mark Hudnall, Marvin Zhang, Marwan Aljubeh, Mateusz Litwin, Matthew Zeng, Max Johnson, Maya Shetty, Mayank Gupta, Meghan Shah, Mehmet Yatbaz, Meng Jia Yang, Mengchao Zhong, Mia Glaese, Mianna Chen, Michael Janner, Michael Lampe, Michael Petrov, Michael Wu, Michele Wang, Michelle Fradin, Michelle Pokrass, Miguel Castro, Miguel Oom Temudo de Castro, Mikhail Pavlov, Miles Brundage, Miles Wang, Minal Khan, Mira Murati, Mo Bavarian, Molly Lin, Murat Yesildal, Nacho Soto, Natalia Gimelshein, Natalie Cone, Natalie Staudacher, Natalie Summers, Natan LaFontaine, Neil Chowdhury, Nick Ryder, Nick Stathas, Nick Turley, Nik Tezak, Niko Felix, Nithanth Kudige, Nitish Keskar, Noah Deutsch, Noel Bundick, Nora Puckett, Ofir Nachum, Ola Okelola, Oleg Boiko, Oleg Murk, Oliver Jaffe, Olivia Watkins, Olivier Godement, Owen Campbell-Moore, Patrick Chao, Paul McMillan, Pavel Belov, Peng Su, Peter Bak, Peter Bakkum, Peter Deng, Peter Dolan, Peter Hoeschele, Peter Welinder, Phil Tillet, Philip Pronin, Philippe Tillet, Prafulla Dhariwal, Qiming Yuan, Rachel Dias, Rachel Lim, Rahul Arora, Rajan Troll, Randall Lin, Rapha Gontijo Lopes, Raul Puri, Reah Miyara, Reimar Leike, Renaud Gaubert, Reza Zamani, Ricky Wang, Rob Donnelly, Rob Honsby, Rocky Smith, Rohan Sahai, Rohit Ramchandani, Romain Huet, Rory Carmichael, Rowan Zellers, Roy Chen, Ruby Chen, Ruslan Nigmatullin, Ryan Cheu, Saachi Jain, Sam Altman, Sam Schoenholz, Sam Toizer, Samuel Miserendino, Sandhini Agarwal, Sara Culver, Scott Ethersmith, Scott Gray, Sean Grove, Sean Metzger, Shamez Hermani, Shantanu Jain, Shengjia Zhao, Sherwin Wu, Shino Jomoto, Shirong Wu, Shuaiqi, Xia, Sonia Phene, Spencer Papay, Srinivas Narayanan, Steve Coffey, Steve Lee, Stewart Hall, Suchir Balaji, Tal Broda, Tal Stramer, Tao Xu, Tarun Gogineni, Taya Christianson, Ted Sanders, Tejal Patwardhan, Thomas Cunninghman, Thomas Degry, Thomas Dimson, Thomas Raoux, Thomas Shadwell, Tianhao Zheng, Todd Underwood, Todor Markov, Toki Sherbakov, Tom Rubin, Tom Stasi, Tomer Kaftan, Tristan Heywood, Troy Peterson, Tyce Walters, Tyna Eloundou, Valerie Qi, Veit Moeller, Vinnie Monaco, Vishal Kuo, Vlad Fomenko, Wayne Chang, Weiyi Zheng, Wenda Zhou, Wesam Manassra, Will Sheu, Wojciech Zaremba, Yash Patil, Yilei Qian, Yongjik Kim, Youlong Cheng, Yu Zhang, Yuchen He, Yuchen Zhang, Yujia Jin, Yunxing Dai, and Yury Malkov. Gpt-4o system card, 2024. URL `https://arxiv.org/abs/2410.21276`.

Joon Sung Park, Joseph C O'Brien, Carrie J Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. Generative agents: Interactive simulacra of human behavior. *arXiv preprint arXiv:2304.03442*, 2023.

Chen Qian, Xin Cong Wang, Yufan Zheng, Cheng Wang, Yequan Cen, Weize Wang, et al. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

David Rein, Ansh Raichur, Caleb Riddoch, Andrew Andreassen, Ben Jones, Zihui Wu, Shufan Jiang, Kevin Chen, Cong Jiang, Andy Zhao, Lucy Yuan, Jerry Li, Yaofeng Zhang, R Arjun Gopalakrishnan, Andrew Pan, Yapei Zhou, Leon Tang, Thomas Lee, Tom Brown, and Jacob Steinhardt. GPQA: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. Rethinking the bounds of llm reasoning: Are multi-agent discussions the key? In *arXiv preprint arXiv:2402.18272*, 2024. Accessed: YYYY-MM-DD.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, March 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*, 2023.

Bohan Yao and Vikas Yadav. A toolbox, not a hammer–multi-tag: Scaling math reasoning with multi-tool aggregation. *arXiv preprint arXiv:2507.18973*, 2025.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations (ICLR)*, 2025. Oral Presentation (Top 1.8%).

A ARM Search Algorithm

Algorithm 1 provides the full pseudocode of the reflection-guided search algorithm for evolving ARM modules.

Algorithm 1 Reflection-Guided Search

Input: Initial program p_{root} (e.g., m_{CoT} or π_{Rec}), evaluation function EVALUATE$(\cdot)$, total iterations K, exploration constant C.
Initialize:
Tree $\mathcal{T}$ with a single node for p_{root}.
$p_{root}.\bar{\mathcal{R}} \leftarrow$ EVALUATE(p_{root}) ▷ Evaluate the baseline program on a validation batch
$p_{root}.N \leftarrow 1$ ▷ Initialize visit count for the root
for $t = 1$ **to** K **do**
▷ *1. Select a parent program to mutate*
$P(p_i) \leftarrow \frac{\exp\left(p_i.\bar{\mathcal{R}}/T\right)}{\sum_{j \in \mathcal{T}} \exp\left(p_j.\bar{\mathcal{R}}/T\right)}$
$p_{parent} \leftarrow$ Sample$(\mathcal{T}, P)$
▷ *2. Expand the tree via reflection*
traces $\leftarrow$ EXECUTE(p_{parent}) ▷ Collect execution traces
history $\leftarrow$ GETMUTATIONHISTORY(p_{parent})
$p_{new} \leftarrow$ REVIEWERAGENT$(p_{parent}, \text{traces}, \text{history})$
▷ *3. Evaluate the new program*
$p_{new}.\bar{\mathcal{R}} \leftarrow$ EVALUATE(p_{new})
$p_{new}.N \leftarrow 1$
▷ *4. Update tree and statistics*
$\mathcal{T}$.ADDCHILD(p_{parent}, p_{new})
$p_{parent}.N \leftarrow p_{parent}.N + 1$
end for
return $\underset{p_i \in \mathcal{T}}{\text{argmax}}\ (p_i.\bar{\mathcal{R}})$ ▷ Return the program with the highest empirical reward

B Theoretical Analysis

A complete theoretical analysis of the multi-agentic system ARM powered by LLMs is intractable due to the complex, high-dimensional nature of language generation and the non-stationary of the generation process. Recent research (Chang et al., 2025; Kim et al., 2025) models sequential CoT reasoning steps as a Markov Decision Process by abstracting away the underlying complexities of the text generation process and focusing on higher level reasoning states. Therefore, to build a formal intuition for the design choices in our scaffolded search for the step-generator, and the decoupled search for the meta-policy (Algorithm1), we also analyze an idealized formulation of the problem as a Markov Decision Process (MDP).

Our analysis is particularly inspired by recent work on self taught reasoners (RL-Star) by Chang et al. (2025), where they introduce a step indexed competence parameter $\delta_{t,n}$ which quantifies the advantage in probability of a correct reasoning step at step n during training iteration t over a baseline random reasoner. They show the conditions under which a bootstrapped RL learning algorithm based on rejection sampling shows monotonic improvement and convergence. While our goals are similar (improving the reasoning process), our problem statement has critical differences which makes a straight forward adaption infeasible: RL-Star analyses a system where the LLM's parametric weights are updated via reinforcement learning. On the other hand, ARM treats the LLM as a black box and performs discrete, evolutionary search (Fernando et al., 2024; Agrawal et al., 2025) over programming modules that orchestrate calls to the LLM. Consequently, our search is inherently *discrete*, so smoothness-based guarantees do not apply. Hence we do not assume or prove convergence guarantees, and instead motivate the intuition of our scaffolded search process as a conservative policy improvement (CPI) (Kakade & Langford, 2002) that preferentially selects modules with higher competence leading to improved reasoning process.

B.1 An Idealized MDP Model of Step-wise Reasoning

We model the reasoning process as a Markov decision process (MDP) Sutton & Barto (2018) $\mathcal{M} = (S, N, A, P, R, \gamma)$:

- **State Space** (S)**:** The state space $S = \mathcal{U} \cup \mathcal{G} \cup \mathcal{F}$ is partitioned into three disjoint subsets:
  - $\mathcal{U}$: A state $s \in \mathcal{U}$ represents a partial reasoning trace $q, p_1, ...p_k$ that is not yet terminated.
  - $\mathcal{G}$: A state $s \in \mathcal{G}$ represents a reasoning path that has successfully ended on the right answer. In our setting this is when the module emits the `/boxed{correct answer}`. This is an absorbing region.
  - $\mathcal{F}$: A state $s \in \mathcal{F}$ represents a reasoning path that has terminated at the wrong answer. In our setting this is when the module emits the `/boxed{incorrect answer}`. This is an absorbing region.
- **Verification Predicate (`solved`):** A predicate function $\mathcal{S} \rightarrow 0, 1$ judging if the right answer is already derivable from the given partial reasoning state. Note that this is a simple formatting action, and is independent of the module m.
  - `solved(s) = 0` $\forall s \in \mathcal{F}$
  - `solved(s) = 1` $\forall s \in \mathcal{G}$
- **Maximum Reasoning Steps (N)**: We rollout the reasoning process up to N steps. After N steps of reasoning, we enforce a *model-independent* termination rule where the state deterministically goes into $s' \in \mathcal{G}$ if `solved(s)` = 1 and into $s' \in \mathcal{F}$ if `solved(s)` = 0. For simplicity of notation, we assume the total trajectory length to be $N + 1$.
- **Action Space** (A): For a fixed meta-policy π_{Rec} that recursively generates steps until termination (such as the one used by baseline CoT or the ARM-only variant), the meta policy executes a single action at any give state $s \in \mathcal{U}$: i.e., invokes a step-generator module m to produce the next reasoning step. Thus, the action space is a singleton $\mathcal{A} = \{\texttt{generate\_step}\}$. For terminal states $\mathcal{G} \cup \mathcal{F}$, this is a `no-op`. Hence, the choice of the module m fully defines the transition dynamics of the MDP.
- **Reward Function** (R)**:** The one-shot terminal reward is sparse:

$$R(s \rightarrow s') = \begin{cases} 1, & s' \in \mathcal{G}, \\ 0, & \text{otherwise.} \end{cases}$$

- **Transition Dynamics (P)**: We denote the state transition probability $P(.|s, m)$ with the Markov assumption. This simplification is the core foundation for our MDP analysis.
- **Value Function:** For $n \in \{0, \ldots, N\}$, let $V_n^m(s)$ denote the value with n reasoning steps remaining before the formatting step. The Bellman recursion can be written as

$$V_n^m(s) = \begin{cases} 1, & s \in \mathcal{G}, \\ 0, & s \in \mathcal{F}, \\ \{\texttt{solved}(s)\} + \{\neg\texttt{solved}(s)\}\, \mathbb{E}_{s' \sim P_m(\cdot|s)}[V_{n-1}^m(s')], & s \in \mathcal{U},\ n \geq 1, \\ \{\texttt{solved}(s)\}, & s \in \mathcal{U},\ n = 0. \end{cases}$$

Within this MDP framework, the ideal objective is to discover a module m^*, that maximizes the expected value from the initial state distribution $d_0(s)$

$$m^* = \arg\max_{m \in \mathcal{M}} \mathbb{E}_{s_0 \sim d_0(s)} \left[V_N^m(s_0)\right]$$

This objective poses several major optimization challenges: 1) credit assignment problem over long sequence of steps and 2) unconstrained search space of code modules.

B.2 Definitions

We introduce the following quantities to characterize module's performance and search strategy:

- **Per-step competence $\delta_m(s)$:** This represents the *competence* of the module m at a reasoning state $s \in \mathcal{U}$ analogous to $\delta_{t,n}$ term in Chang et al. (2025). The probability that a one step update $s \in \mathcal{U}$ is valid can be viewed as a monotonically increasing function over $\delta_m(.)$, but for simplicity of notation, we assume this to be $\delta_m(s)$ itself.
- **Recovery $r_m(s)$:** The probability that, conditioned on an invalid one-step update from s, the next step returns to a valid state. This term captures the recovery possibility of a mistake in the immediate next turn. While recovery can happen at any turn following the mistake in a real LLM, we limit the window to 1 for simplicity.
- **Composite Validity $\phi_m(s)$:** The total probability that the next step is valid for a step $s \in \mathcal{U}$, either by being immediately valid, or by being successfully repaired on the next step:

$$\phi_m(s) = \delta_m(s) + (1 - \delta_m(s))r_m(s) \tag{1}$$

- **Window $W = (n, l)$:** A block of l consecutive steps starting at step index n where all states remain in $\mathcal{U}$. The ARM module replaces the baseline module m_{CoT} with a candidate module m only on this window.
- **Visitation Weights $w^\pi(W)$:** The probability under baseline policy (π, m_{CoT}) that the window W occurs. This measures the frequency with which the meta-policy starts the module at a given window.

B.3 Key Simplifying Assumptions

The rest of our analysis relies on the following key assumptions.

Assumption 1. [Local competence lift in the scaffolding window] Within any given window $W = (n, l)$, for all states s visited, the candidate model satisfies $\phi_m(s) \geq \phi_{m_{CoT}(s)} + \Delta_c$ for some lift $\Delta_c \in [0, 1)$.

Rationale: This is the empirical premise that our scaffolded search objective (Section 2) is designed to optimize for. Our Algorithm 1 directly measures and selects for modules that improve local validity and recovery rates over the CoT within a constrained context at random locations.

Assumption 2. [Compatibility Loss] We define $\beta_l(W)$ a bound on the probability that replacing the baseline m_{CoT} with ARM module m at a window W yields a context which is unusable for the rest of the baseline reasoning trace. We refer to $(1 - \beta_l(W))$ as the *compatibility factor*. Furthermore, we define $\overline{\beta}_l := \sup_{W \in \mathcal{W}_{\text{valid}}} \beta_l(W)$ as the supremum of incompatibility probabilities across all valid windows, representing the worst-case incompatibility bound.

Rationale: Swapping the baseline module m_{CoT} with m can introduce a "*context drift*" or "*semantic drift*" which could amplify at deployment time when the ARM module m is used through the entire trajectory. Our approach minimizes this drift by two means: 1) the few shot examples of the progress, as well as the provided partial progress acts as a powerful inductive bias to constrain the next step to states that preserve usefulness, for example by adopting the same notation, logical continuity, etc. 2) the reviewer agent which proposes mutations to the module (starting from baseline CoT) is prompted to generate modules which solve one step at a time starting from the given partial progress.

Assumption 3. [Module-invariant termination] We assume that the reward is terminal, and is provided under the condition that the extracted answer matches the right answer. Furthermore, the last step in the MDP is reserved for this extraction, which is considered module-invariant, i.e. both CoT or any other module can do this final syntax step (equally) perfectly.

B.4 Theoretical Grounding for the Scaffolded Step-Generator Search

The scaffolded objective evaluates a candidate m by *splicing* it into a baseline rollout for a short window $t \in \{i, \ldots, i + \ell - 1\}$ while keeping m_{CoT} before and after:

$$\underbrace{U^*_{m_{\text{CoT}}} \circ \left(U^\ell_m\right) \circ U^i_{m_{\text{CoT}}}}_{\text{"baseline–candidate–baseline"}}.$$

This section formalizes the link between local module improvements and global performance gains. Under our simplified MDP framework and assumptions 1, 2, we establish the following lemmas:

Lemma 1. [Window lift from local competence] The probability of remaining in $\mathcal{U}$ after the window increases by at least $l.C^{l-1}.\Delta_c$ for some constant $C \in (0, 1)$.

Proof: The l step window survival (being in $\mathcal{U}$) is at least $\phi_m(s)^l$. Under Assumption 1, the per-step composite validity improves by at least Δ_c. Hence, the survival probability improves by at least $\left(\phi_{\mathrm{CoT}}(s) + \Delta_c\right)^l - \phi_{\mathrm{CoT}}(s)^l$. Applying the mean value theorem on $f(x) = x^\ell$, we get

$$(x + \Delta_c)^\ell - x^\ell = \ell\xi^{\ell-1}\Delta_c \geq \ell(\min_{s\in\mathcal{U}_\mathcal{R}} \phi_{\mathrm{CoT}}(s))^{\ell-1}\Delta$$

for some $\xi \in [x, x + \Delta]$, where $\mathcal{U}_R$ represents states reachable by baseline CoT module. Further, let $C := \min_{s\in U_R} \phi_{\mathrm{CoT}}(s)$ be a constant.

Lemma 2. [Accounting for Compatibility] The probability of a sample surviving the window W and remain usable after the module swap is lower bounded by $(1 - \beta_l(W)).(l.C^{l-1}.\Delta_c)$ where C is the constant from Lemma 1.

Proof: From Assumption 2, the usability probability is at least $(1 - \beta_l(W))$. Multiplying this by the probability of surviving the window from Lemma 1 yields the result.

Lemma 3. [From window survival to finite-horizon success] Any increase in the probability of staying within $\mathcal{U}$ across a window W (while remaining usable) weakly increases the probability of reaching $\mathcal{G}$ within the horizon N.

Proof: Under assumption 3, the termination rule is module-invariant, and reaching the goal state only depends on being in state $s \in \mathcal{U} \cup \mathcal{G}$ and `solved(s)=1`. Thus a higher probability of preserving valid, in-progress states across a window cannot decrease (and generally increases) the likelihood that subsequent steps will generate such a solvable state before the horizon is exhausted. This follows from standard monotonicity arguments on absorbing Markov chains.

Theorem 1 (Gain from Scaffolded Module Substitution in recursive meta policy)**.** Let $J(\pi_{Rec}, m) = \mathbb{E}\left[V_N^{\pi_{Rec},m}(s_0)\right]$ denote the expected terminal reward (success probability) obtained when recursively applying the step-generator module m under a fixed baseline meta-policy π_{Rec} for horizon N. The improvement of the ARM module m^* over m_{CoT} is at least:

$$J(\pi_{Rec}, m^*) - J(\pi_{Rec}, m_{\mathrm{CoT}}) \;\geq\; \sum_W w_\pi(W)\,\kappa(W)\,(1 - \beta_l(W))\,l\,C^{l-1}\,\Delta_c. \tag{2}$$

where each term represents:

- $w_\pi(W)$: visitation probability of a window W under a baseline rollout;
- $\kappa(W)$: probability that a usable post-window state leads to terminal success within the remaining horizon.
- $C \in (0, 1]$: a constant from Lemma 1 capturing compounding survival over steps.

In particular, if $\kappa(W) \geq \kappa_{min} \geq 0$ for all W, then

$$J(\pi_{Rec}, m^*) - J(\pi_{Rec}, m_{\mathrm{CoT}}) \;\geq\; \kappa_{min}\,(1 - \beta_l(W))\,l\,C^{l-1} \sum_W w_\pi(W) \tag{3}$$

Proof: From Lemma 2 , the probability of surviving the window is lower bounded by $(1 - \beta_l)lC^{l-1}$. Let $\kappa(W)$ represent the probability of success upon starting from a good state, post the window. By Lemma 3, the increase in usable post-window mass translates to atleast a $\kappa(W)$ fraction improvement in terminal success within the remaining horizon. Thus the expected gain from the window is $\kappa(W)(1 - \beta_l(W))lC^{l-1}$. Taking expectation over window visitation probabilities yields the result:

$$\begin{aligned} J(\pi_{Rec}, m^*) - J(\pi_{Rec}, m_{\mathrm{CoT}}) &= \sum_W w_\pi(W) \cdot \mathrm{Gain}(W) \\ &= \sum_W w_\pi(W) \cdot \kappa(W)\,(1 - \beta_l(W))\,l\,C^{l-1}. \\ &\geq \sum_W w_\pi(W) \cdot \kappa_{min}\,(1 - \beta_l(W))\,l\,C^{l-1}. \end{aligned} \tag{4}$$

where $k_{min} \geq 0$ is the lowest probability of success from a valid, usable intermediate reasoning trace.

B.5 Theoretical Justification for zero-shot policy transfer

The learned meta policy π^* uses m_{CoT} as the *step generator* during the learning phase and is deployed zero-shot using the discovered ARM module m^*. Below, we justify why this transfer is effective.

Theorem 2 (Validity of Zero-shot step generator swap in Meta policy)**.** Let $J(\pi^*, m^*) = \mathbb{E}_{s_0 \sim \mathcal{D}}\left[V_N^{(\pi^*, m^*)}(s_0)\right]$ denote the expected terminal reward obtained when applying the discovered meta-policy π^* (from Section 2), with the step-generator module m^* under horizon N. If $\Delta_c \geq \frac{\overline{\beta_l}}{1-\overline{\beta_l}}$, then the transfer is valid, i.e., $J(\pi^*, m^*) \geq J(\pi^*, m_{CoT})$

Let's define per-step advantage of a module m over m_{CoT} with n more steps to go as the expected difference in value when taking one step with m and the rest with m_{CoT}:

$$A_n(s_n, m) \triangleq \mathbb{E}_{s' \sim P_m(.|s)}[V_{n-1}^{(\pi^*, m_{CoT})}(s')] - \mathbb{E}_{s' \sim P_{m_{CoT}}(.|s)}[V_{n-1}^{(\pi^*, m_{CoT})}(s')] \tag{5}$$

Now let's consider the difference in expected value starting from a given state s_0 sampled from the data distribution $\mathcal{D}$. For simplicity, we drop π^* from notation as it is the common meta policy in both terms.

$$V_N^m(s_0) - V_N^{m_{CoT}}(s_0)$$

Rolling out for one step yields

$$\mathbb{E}_{s_1 \sim P_m(.|s_0)}[V_{n-1}^m(s_1)] - V_n^{m_{CoT}}(s_0)$$

Adding an subtracting $\mathbb{E}_{s_1 \sim P_m(.|s_0)}[V_{n-1}^{m_{CoT}}(s_1)]$ (i.e., sampling from m but continue with m_{CoT}) we get:

$$\begin{aligned} \mathbb{E}_{s_1 \sim P_m(\cdot|s_0)}[V_{n-1}^{m_{CoT}}(s_1)] - \mathbb{E}_{s_1 \sim P_{m_{CoT}}(\cdot|s_0)}[V_{n-1}^{m_{CoT}}(s_1)] \\ + \mathbb{E}_{s_1 \sim P(\cdot|s_0)}[V_{n-1}^m(s_1) - V_{n-1}^{m_{CoT}}(s_1)] \end{aligned} \tag{6}$$

By Equation-5, this can be written as

$$A_n^{m_{CoT}}(s_0, m) + \mathbb{E}_{s_1 \sim P(.|s_0)}[V_{n-1}^m(s_1) - V_{n-1}^{m_{CoT}}(s_1)]$$

This is a recursive equation in n since the second term is the difference in value between the module with $n-1$ steps to horizon. Hence:

$$V_N^m(s_0) - V_N^{m_{CoT}}(s_0) = \sum_{n=0}^{N} \left[A_{N-n}^{m_{CoT}}(s_n, m)\right] \tag{7}$$

Thus we can conservatively guarantee module improvement, if each of the the advantage term is positive. Suppose that U represents the event that one step rollout using our discovered module m^* is usable (i.e. no errors, and usable context) in the next turn, then by law of total expectation the advantage term can be written as:

$$\begin{aligned} &\mathbb{P}(U \mid s, m^*) \cdot \big(\mathbb{E}[V_{N-n} \mid s, m^*, U] - \mathbb{E}[V_{N-n} \mid s, m_{\text{CoT}}, U]\big) \\ &\quad + \ \mathbb{P}(\neg U \mid s, m^*) \cdot \big(\mathbb{E}[V_{N-n} \mid s, m^*, \neg U] - \mathbb{E}[V_{N-n} \mid s, m_{\text{CoT}}, \neg U]\big) \end{aligned} \tag{8}$$

By Assumption 1 and Assumption 2, the first term is at least $(1 - \overline{\beta_l}) \cdot \Delta_c$. The second term is lower bounded in the worst case by $\overline{\beta_l} \cdot (-1)$ since the probability of non-useful state is $\overline{\beta_l}$ and the difference in reward is at most -1 (when m^* fails where m_{CoT} succeeds). Thus a conservative lower bound is:

$$A_n^{m_{CoT}}(s, m) \geq (1 - \overline{\beta_l}) \cdot \Delta_c - \overline{\beta_l} \tag{9}$$

Hence advantage is non-negative when $\Delta_c \geq \frac{\overline{\beta_l}}{1-\overline{\beta_l}}$, and we can guarantee that zero-shot transfer is effective when this condition is satisfied.

Corrolary 1. The meta policy, paired with ARM, (π^*, m^*) yields the highest performance when Δ_c is large and β_l is small.

Remark: From Equation-8, the claim holds. Our scaffolded objective function is precisely designed to maximize Δ_c while reducing the probability of state drift leading to mismatched compatibility. In 3, we can empirically see that better performance is associated with improvement in per-step competence. And in 2, we see that the ARM module is compatible with m_{CoT} and can act as a superior drop-in replacement for CoT. Furthermore, the examples shown in Section C show that both CoT and ARM rollouts are empirically non-discriminable.

C Step Generator roll out Examples

C.1 Example-1

```
### Step 2
Express the sides in terms of the common ratio \(2R\):
\[
a = 2R \sin 60^\circ = 2R \times \frac{\sqrt{3}}{2} = R \sqrt{3},
\]
\[
b = 2R \sin 45^\circ = 2R \times \frac{\sqrt{2}}{2} = R \sqrt{2},
\]
\[
c = 2R \sin 75^\circ.
\]
Recall that \(\sin 75^\circ = \sin(45^\circ + 30^\circ)\):
\[
\sin 75^\circ = \sin 45^\circ \cos 30^\circ + \cos 45^\circ \sin 30^\
    circ = \frac{\sqrt{2}}{2} \times \frac{\sqrt{3}}{2} + \frac{\sqrt
    {2}}{2} \times \frac{1}{2} = \frac{\sqrt{6}}{4} + \frac{\sqrt
    {2}}{4} = \frac{\sqrt{6} + \sqrt{2}}{4}.
\]
Hence,
\[
c = 2R \times \frac{\sqrt{6} + \sqrt{2}}{4} = R \times \frac{\sqrt{6}
    + \sqrt{2}}{2}.
\]
```

Listing 1: ARM rollout

```
### Step 2
Expressing the sides in terms of the circumradius \( R \):
\[
a = 2R \sin A = 2R \sin 60^\circ = 2R \times \frac{\sqrt{3}}{2} = R \
    sqrt{3},
\]
\[
b = 2R \sin B = 2R \times \frac{\sqrt{2}}{2} = R \sqrt{2},
\]
\[
c = 2R \sin C = 2R \times \sin 75^\circ.
\]
Recall that:
\[
\sin 75^\circ = \sin (45^\circ + 30^\circ) = \sin 45^\circ \cos 30^\
    circ + \cos 45^\circ \sin 30^\circ = \frac{\sqrt{2}}{2} \times \
    frac{\sqrt{3}}{2} + \frac{\sqrt{2}}{2} \times \frac{1}{2} = \frac
    {\sqrt{6}}{4} + \frac{\sqrt{2}}{4} = \frac{\sqrt{6} + \sqrt
    {2}}{4}.
\]
Therefore:
\[
c = 2R \times \frac{\sqrt{6} + \sqrt{2}}{4} = R \times \frac{\sqrt{6}
    + \sqrt{2}}{2}.
```

```
\]
```

Listing 2: CoT rollout

C.2 Example-2

```
### Step 3
Now, determine the new temperature after doubling the kinetic energy:
\[
T_2 = 526.3\,K
\]
Convert this back to Celsius:
\[
T_2^{\circ} C = 526.3 - 273.15 = 253.15^\circ C
\]
This is a significant increase from the initial \(-10^\circ C\),
    indicating the molecules now have enough energy corresponding to
    about \(253^\circ C\).
```

Listing 3: ARM rollout

```
### Step 3
Determine the new temperature after doubling the kinetic energy:
Since kinetic energy \( \propto T \),
\[
T_2 = 2 \times T_1 = 526.3 \, K
\]
Convert this back to Celsius:
\[
T_2 (^\circ C) = T_2 (K) - 273.15 = 526.3 - 273.15 \approx 253.15^\
    circ C
\]
This temperature is well above the boiling point of water at 1 atm
    (100C). Therefore, the molecules' kinetic energy would be
    sufficient to overcome intermolecular forces and convert the water
     from the solid phase (ice) into the gas phase (vapor).
```

Listing 4: CoT rollout

C.3 Example-3

```
### Step 1

To determine the correct substitution from equation 12.9 to equation
    12.10, we need to understand the context of these equations and
    the variables involved. Equation 12.9 likely involves a momentum-
    like variable \( P^+ \) or a gamma matrix that acts on spinors,
    and the goal is to simplify or rewrite this expression into a form
     involving gamma matrices or energy-momentum components.

Typically, in relativistic quantum mechanics or quantum field theory,
    the transformation from the light-cone momentum \( P^+ \) to gamma
     matrices involves choosing a specific gamma matrix to project or
    decompose spinor components. The options provided suggest that the
     substitution involves gamma matrices and possibly the momentum
    operator or gamma matrices related to time.

Given that, the known structures are:

- \( P^+ \): light-cone momentum component, usually \( P^+ = P^0 + P^3
    \).
- \( \gamma^u \): a gamma matrix with a specific index \( u \),
    possibly a spatial index.
- \( \gamma^0 \): the gamma matrix corresponding to time.
```

```
- \( \gamma^0 P \): a product of \( \gamma^0 \) with some momentum
    operator \( P \).

Since the transformation is from equation 12.9 to 12.10, and the
    options involve gamma matrices and momenta, the typical approach
    in such derivations involves rewriting the projection operator or
    the momentum in terms of gamma matrices.
```

Listing 5: ARM rollout

```
### Step 1

To determine the correct substitution, first understand the context:
    equation 12.9 involves the momentum operator \( P^\mu \), and
    equation 12.10 is likely a transformed version involving gamma
    matrices. In relativistic quantum mechanics, transformations often
     involve gamma matrices \( \gamma^\mu \) and the gamma matrices'
    role in formulating Dirac equations.

Given the options:

- A: \( P^+ \)
- B: \( \gamma^u \)
- C: \( \gamma^0 \)
- D: \( \gamma^0 P \)

and based on the typical transformations, the substitution from \( P^\
    mu \) to a form involving gamma matrices usually involves \( \
    gamma^0 \) because it relates to the time component of the gamma
    matrices and appears frequently in the Dirac formalism.

In particular, if the original equation involves \( P^\mu \), and the
    transformed equation involves a gamma matrix, then the key is
    recognizing the common substitution that converts momentum
    operators into gamma matrix expressions, especially the time
    component \( P^0 \).

Therefore, the step is to analyze which gamma matrix substitution
    correctly transforms the operator form in equation 12.9 into the
    one in 12.10.
```

Listing 6: CoT rollout

D Additional Evaluations

To demonstrate the generalizablilty of the discovered ARM module and the meta policy, we evaluate our approach zero-shot on two other benchmarks. For reasoning evaluations on specialized scientific knowledge, we used *GPQA*, a benchmark containing graduate-level questions in physics, chemistry, and biology designed to be challenging even for human experts (Rein et al., 2023). Finally, to measure practical, up-to-date reasoning on Code, Data Analysis, Common Sense Reasoning, and robustness against data contamination, we used *LiveBench Reasoning* [4], a dynamic benchmark with continuously evolving questions (Jain et al., 2024).

Model	Method	GPQA	LiveBench
GPT-4.1-nano	CoT	50.0%	33.1%
	CoT-SC	50.6%	36.9%
	Self-Refine	50.0%	28.1%
	LLM-Debate	52.5%	33.8%
	ADAS (test set)	48.1%	31.2%
	ADAS (1000-sample)	46.8%	29.4%
	AFlow (test set)	39.9%	30.6%
	AFlow (1000-sample)	51.3%	30.6%
	ARM **(Ours)**	<u>60.1%</u>	<u>39.4%</u>
	ARM + MP **(Ours)**	**61.4%**	**45.6%**
GPT-4o	CoT	53.8%	46.2%
	CoT-SC	53.2%	42.5%
	Self-Refine	53.8%	37.5%
	LLM-Debate	56.3%	<u>47.5%</u>
	ADAS (test set)	46.2%	38.8%
	ADAS (1000-sample)	46.8%	41.9%
	AFlow (test set)	53.8%	41.9%
	AFlow (1000-sample)	50.6%	45.0%
	ARM **(Ours)**	<u>59.5%</u>	<u>47.5%</u>
	ARM + MP **(Ours)**	**60.1%**	**51.9%**
LLaMA-3.3-70B	CoT	50.0%	38.1%
	CoT-SC	**53.2%**	45.0%
	Self-Refine	<u>51.3%</u>	<u>46.9%</u>
	LLM-Debate	50.6%	46.2%
	ADAS (test set)	47.5%	37.5%
	ADAS (1000-sample)	42.4%	46.2%
	AFlow (test set)	46.8%	38.1%
	AFlow (1000-sample)	46.8%	15.6%
	ARM **(Ours)**	49.6%	46.2%
	ARM + MP **(Ours)**	50.0%	**50.0%**

Table 2: Additional results on two complex reasoning benchmarks in analytical domains across three foundation models. We compare against two groups of baselines: (1) foundational reasoning strategies used to build agentic systems (CoT, CoT-SC, Self-Refine, and LLM-Debate), and (2) existing state-of-the-art automatic MAS design methods (ADAS and AFlow). Our approach is presented in two variants: **ARM**, which recursively applies the discovered reasoning module, and our full method, **ARM + MP**, which combines the ARM with a learned Meta-Policy (MP). Best score in each category is **bolded** and second best score is <u>underlined</u>.

[4] `https://huggingface.co/datasets/livebench/reasoning`

E Best ARM discovered: CriticChainOfThoughtV7

The following is the Python implementation of the best ARM discovered by our algorithm.

```python
import asyncio

class CriticChainOfThoughtV7:
    def __init__(self, llm):
        self.llm = llm

    async def forward(self, problem, partial_progress):
        # 1. Generate four candidate next steps in parallel
        candidate_tasks = [
            self.llm.generate_step(problem, partial_progress)
            for _ in range(4)
        ]
        candidates = await asyncio.gather(*candidate_tasks)

        # 2. Critique candidates in two groups of two, in parallel
        critique_tasks = []
        groups = [
            (0, 2, ("rating_1", "rating_2"), ("critique_1", "critique_2")),
            (2, 4, ("rating_3", "rating_4"), ("critique_3", "critique_4"))
        ]
        for start, end, rating_names, critique_names in groups:
            context = [
                {
                    "name": "Problem",
                    "data": problem,
                    "description": "The problem to solve."
                },
                {
                    "name": "Partial Progress",
                    "data": partial_progress,
                    "description": "The partial solution so far."
                },
                {
                    "name": "Candidate Next Steps",
                    "data": "\n\n".join(
                        f"### Candidate Next Step {i+1}\n{candidates[i]}"
                        for i in range(start, end)
                    ),
                    "description": "Two candidate next steps formatted with markdown subheaders."
                }
            ]
            instructions = (
                "You are given a problem, the current partial solution, and two candidate next reasoning steps.\n"
                "For each candidate, provide:\n"
                f"- {rating_names[0]} and {rating_names[1]}: a single integer rating from 1 to 10 indicating its fit as the next reasoning step (10 is best).\n"
                f"- {critique_names[0]} and {critique_names[1]}: a one-sentence critique highlighting each candidate's strengths and weaknesses.\n"
                f"Name the fields exactly {rating_names[0]}, {critique_names[0]}, {rating_names[1]}, {critique_names[1]}."
            )
            response_format = [
                {
                    "name": rating_names[0],
```

```
                    "description": f"Integer rating (1-10) for Candidate Next Step {start+1}."
                },
                {
                    "name": critique_names[0],
                    "description": f"One-sentence critique of Candidate Next Step {start+1}."
                },
                {
                    "name": rating_names[1],
                    "description": f"Integer rating (1-10) for Candidate Next Step {start+2}."
                },
                {
                    "name": critique_names[1],
                    "description": f"One-sentence critique of Candidate Next Step {start+2}."
                }
            ]
            critique_tasks.append(
                self.llm.chat_completion(context, instructions, response_format)
            )

        critiques = await asyncio.gather(*critique_tasks)

        # 3. Parse ratings and identify the two highest-rated candidates
        ratings = [
            int(critiques[0]["rating_1"]),
            int(critiques[0]["rating_2"]),
            int(critiques[1]["rating_3"]),
            int(critiques[1]["rating_4"])
        ]
        sorted_indices = sorted(range(4), key=lambda i: ratings[i], reverse=True)
        top1_idx, top2_idx = sorted_indices[0], sorted_indices[1]
        top1_candidate = candidates[top1_idx]
        top2_candidate = candidates[top2_idx]

        # 4. Final head-to-head comparison between the top two candidates
        context_final = [
            {
                "name": "Problem",
                "data": problem,
                "description": "The problem to solve."
            },
            {
                "name": "Partial Progress",
                "data": partial_progress,
                "description": "The partial solution so far."
            },
            {
                "name": "Candidate Next Steps",
                "data": (
                    f"### Candidate A\n{top1_candidate}\n\n"
                    f"### Candidate B\n{top2_candidate}"
                ),
                "description": "Two top candidate next steps formatted with markdown subheaders."
            }
        ]
        instructions_final = (
```

```
            "Compare Candidate A and Candidate B as the next reasoning step for the given problem and partial progress.\n"
            "Provide:\n"
            "- winner: choose either 'Candidate A' or 'Candidate B' indicating which step is better.\n"
            "- justification: one-sentence justification for your choice."
        )
        response_format_final = [
            {
                "name": "winner",
                "description": "Either 'Candidate A' or 'Candidate B' indicating the better next step."
            },
            {
                "name": "justification",
                "description": "One-sentence justification for the choice."
            }
        ]
        final_decision = await self.llm.chat_completion(
            context_final, instructions_final, response_format_final
        )

        if final_decision["winner"].strip() == "Candidate A":
            selected_candidate = top1_candidate
            runnerup_candidate = top2_candidate
        else:
            selected_candidate = top2_candidate
            runnerup_candidate = top1_candidate

        # 5. Post-selection adversarial critique with severity rating
        context_flaw = [
            {
                "name": "Problem",
                "data": problem,
                "description": "The problem to solve."
            },
            {
                "name": "Partial Progress",
                "data": partial_progress,
                "description": "The partial solution so far."
            },
            {
                "name": "Selected Candidate Next Step",
                "data": f"### Selected Candidate Next Step\n{selected_candidate}",
                "description": "The final chosen candidate next reasoning step formatted with a markdown subheader."
            }
        ]
        instructions_flaw = (
            "You are given a problem, the current partial solution, and a selected next reasoning step.\n"
            "Identify any major flaw or missing piece of reasoning in the selected step.\n"
            "Provide:\n"
            "- flaw: either the single word 'None' if there is no flaw, or a brief description of the flaw.\n"
            "- severity: a single integer rating from 1 to 10 indicating how severe the flaw is (10 is critical)."
        )
        response_format_flaw = [
            {
                "name": "flaw",
```

```
                "description": "Either the single word 'None' if there is no flaw, or a brief description of a major flaw in the selected step."
            },
            {
                "name": "severity",
                "description": "Integer rating (1-10) indicating severity of the flaw (10 is most severe)."
            }
        ]
        flaw_response = await self.llm.chat_completion(
            context_flaw, instructions_flaw, response_format_flaw
        )
        flaw = flaw_response["flaw"].strip()
        severity = int(flaw_response["severity"])

        # 6. Compute dynamic severity threshold based on rating gap
        gap = ratings[top1_idx] - ratings[top2_idx]
        if gap <= 1:
            threshold = 5
        elif gap == 2:
            threshold = 6
        else:
            threshold = 7

        # 7. If a severe flaw is detected above the dynamic threshold, fall back
        if flaw.lower() != "none" and severity >= threshold:
            return runnerup_candidate
        return selected_candidate
```

Listing 7: "Code for CriticChainOfThoughtV7 with performance 38.0"

F Best Meta-Policy Discovered: VerifiedWeightedAdaptiveSelfConsistentChainOfThought

The following is the Python implementation of the best meta-policy discovered by our algorithm.

```
import asyncio
from agent.solution import Solution, Step
from judge_utils import judge_equality

class VerifiedWeightedAdaptiveSelfConsistentChainOfThought:
    def __init__(self, llm, block):
        self.llm = llm
        self.block = block

    async def forward(self, problem):
        # Helper: generate one chain up to 8 steps, then complete via LLM if needed
        async def generate_chain():
            solution = Solution()
            for _ in range(8):
                next_step = await self.block.forward(problem, str(solution))
                solution.add_step(Step(str(next_step)))
                if solution.is_completed():
                    return solution
            completion = await self.llm.complete_solution(problem, str(solution))
            solution.add_step(Step(str(completion)))
            return solution

        # Helper: confidence scoring (1-5)
```

```
        async def score_chain(chain):
            context = [
                {"name": "Problem", "data": problem, "description": "The original problem statement."},
                {"name": "Chain",   "data": str(chain), "description": "Full chain-of-thought reasoning plus final answer."}
            ]
            instructions = (
                "You are evaluating the chain-of-thought solution for the given problem. "
                "On a scale from 1 (very uncertain) to 5 (very confident), rate your confidence "
                "that the final answer is correct. Output ONLY the integer confidence (1-5)."
            )
            response_format = [{"name": "Confidence", "description": "Integer from 1 to 5"}]
            resp = await self.llm.chat_completion(context, instructions, response_format)
            # parse safely
            try:
                conf = int(resp["Confidence"].strip())
            except Exception:
                conf = 1
            return max(1, min(conf, 5))

        # Helper: verify logical consistency (Yes/No)
        async def verify_chain(chain):
            context = [
                {"name": "Problem", "data": problem, "description": "The original problem statement."},
                {"name": "Chain",   "data": str(chain), "description": "Full chain-of-thought reasoning plus final answer."}
            ]
            instructions = (
                "Review the chain-of-thought reasoning for the given problem. "
                "Is the reasoning free of logical errors or contradictions? "
                "Output ONLY 'Yes' if it is fully logical, otherwise output 'No'."
            )
            response_format = [{"name": "Valid", "description": "Yes or No"}]
            resp = await self.llm.chat_completion(context, instructions, response_format)
            valid = resp.get("Valid", "").strip().lower().startswith("y")
            return valid

        # Weighted vote helper
        def find_best_weighted(chains_list, conf_list):
            weight_sums = {}
            total = sum(conf_list)
            for chain, cf in zip(chains_list, conf_list):
                ans = chain.answer()
                weight_sums[ans] = weight_sums.get(ans, 0) + cf
            best_ans, best_w = None, -1
            for ans, w in weight_sums.items():
                if w > best_w:
                    best_ans, best_w = ans, w
            return best_ans, best_w, total

        # 1) Generate initial 3 chains in parallel
        initial = [generate_chain() for _ in range(3)]
```

```
        chains = await asyncio.gather(*initial)

        # 2) Score and verify each chain
        score_tasks = [score_chain(ch) for ch in chains]
        verify_tasks = [verify_chain(ch) for ch in chains]
        confidences = await asyncio.gather(*score_tasks)
        valids = await asyncio.gather(*verify_tasks)

        max_chains = 7

        # 3) Adaptive sampling with verification gating
        while True:
            # Determine which chains to consider: only verified if any, else all
            if any(valids):
                considered_chains = [ch for ch, v in zip(chains, valids) if v]
                considered_confs   = [cf for cf, v in zip(confidences, valids) if v]
            else:
                considered_chains = chains
                considered_confs   = confidences

            best_ans, best_weight, total_weight = find_best_weighted(considered_chains, considered_confs)
            # stop if weighted majority reached or chain cap
            if best_weight > total_weight / 2 or len(chains) >= max_chains:
                break

            # else generate one more chain, score & verify, then loop
            new_chain = await generate_chain()
            chains.append(new_chain)
            new_conf = await score_chain(new_chain)
            confidences.append(new_conf)
            new_valid = await verify_chain(new_chain)
            valids.append(new_valid)

        # 4) Select final chain: consensus & highest confidence among considered
        if any(valids):
            final_pool = [ (ch, cf) for ch, cf, v in zip(chains, confidences, valids) if v and judge_equality(ch.answer(), best_ans) ]
        else:
            final_pool = [ (ch, cf) for ch, cf in zip(chains, confidences) if judge_equality(ch.answer(), best_ans) ]

        selected_chain = None
        top_conf = -1
        for ch, cf in final_pool:
            if cf > top_conf:
                selected_chain, top_conf = ch, cf

        # Fallback if nothing selected
        if selected_chain is None:
            selected_chain = chains[-1]

        return selected_chain
```

Listing 8: "Code for VerifiedWeightedAdaptiveSelfConsistentChainOfThought with performance 42.0"

G Reproducibility Statement

Upon publication, we commit to releasing the open-source code for our framework, including all discovered Agentic Reasoning Modules, meta-policies, and the specific prompts used for the Reviewer Agent. Our experiments were conducted using a mix of closed and open-source models. The MAS designer utilized o4-mini-high. The reasoning modules were executed on GPT-4.1-nano, GPT-4o, and the open-source Llama-3.3-70B. All evaluation benchmarks, including MATH500, AIME, and HMMT, are publicly available.

G.1 Baseline Implementation Details

As in the ARM implementation, whenever sampling from the MAS executor model, we use a temperature of 1.0 with a top_p of 0.95.

- CoT: We use a simple CoT prompt that instructs the model to reason step-by-step and follow the final answer format.
- CoT-SC: We use $n = 12$ parallel reasoning traces.
- Self-Refine: We limit to a maximum of 5 self refining iterations.
- LLM-Debate: We use 4 LLM agents debating for a maximum of 3 rounds.
- ADAS: We use the provided codebase, following the recommended run configuration. For a fair comparison to other baselines, we make a one line addition to the optimizer prompt to disallow arbitrary Python code execution within the discovered MASes, since other baselines do not utilize code execution. For the 1000-sample optimization, we use GPT-4.1-nano as the MAS executor model during optimization, following ARM's implementation.
- AFlow: We use the provided codebase, following the recommended run configuration. We allow the optimizer to utilize the Custom, AnswerGenerate, and ScEnsemble operators. For the 1000-sample optimization, we use GPT-4.1-nano as the MAS executor model during optimization, following ARM's implementation.