

# ARM: DISCOVERING AGENTIC REASONING MODULES FOR GENERALIZABLE MULTI-AGENT SYSTEMS

**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 various complex reasoning tasks. Recent works have proposed techniques to automate the design of MASes, eliminating the need for manual engineering. However, these techniques perform poorly, often achieving similar or inferior performance to simple baselines. Furthermore, they require computationally expensive re-discovery of architectures for each new task domain and expensive data annotation on domains without existing labeled validation sets. A critical insight is that simple Chain of Thought (CoT) reasoning often performs competitively with these complex systems, suggesting that the fundamental reasoning unit of MASes, CoT, 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 **Agentic Reasoning Module** (ARM), an agentic generalization of CoT where each granular reasoning step is executed by a specialized reasoning module. This module is discovered through a tree search over the code space, starting from a simple CoT module and evolved using mutations informed by reflection on execution traces. The resulting ARM acts as a versatile reasoning building block which can be utilized as a direct recursive loop or as a subroutine in a learned meta-orchestrator. Our approach significantly outperforms both manually designed MASes and state-of-the-art automatic MAS design methods. Crucially, MASes built with ARM exhibit superb generalization, maintaining high performance across different foundation models and task domains without further optimization.

## 1 INTRODUCTION

Chain-of-thought (CoT) prompting has emerged as one of the most effective techniques for eliciting complex reasoning from Large Language Models (LLMs) (Wei et al., 2022). By instructing models to generate a series of intermediate steps that lead to a final answer, CoT significantly enhances performance on tasks requiring arithmetic, commonsense, and symbolic reasoning (Nye et al., 2021; Kojima et al., 2022). This simple yet powerful method allows LLMs to break down complex problems into more manageable sub-problems, effectively externalizing the reasoning process over a sequence of generated tokens before arriving at a solution (Wei et al., 2022; Yao et al., 2023a). Recent advancements have also extended CoT with formal verification and multi-agent perspectives, such as MA-Lot (Wang et al., 2025).

Building on the capabilities of individual LLMs, Multi-Agent Systems (MAS) have recently achieved state-of-the-art results on complex reasoning benchmarks (Park et al., 2023; Qian et al., 2023; Hong et al., 2023). These systems typically consist of multiple LLM-powered agents, each assigned a specific role or expertise, orchestrated by a meta-agent or a predefined communication protocol (Wu et al., 2023; Li et al., 2023). While the collaborative nature of MAS enables division

---

\*Work done during internship at ServiceNow

of labor and synthesis of diverse perspectives (Chen et al., 2023; Dong et al., 2023), recent work has shifted toward the automatic construction of such systems. Emerging automatic MAS generation frameworks demonstrate how agent roles, communication protocols, and workflows can be synthesized directly by LLMs without manual design. For instance, FlowReasoner and AFlow illustrate this trend by automatically generating agent roles and workflows for LLM-based systems, reducing the need for manual design (Zhang et al., 2025c; Kim et al., 2024).

Although MAS approaches have consistently pushed the boundaries of performance, recent studies have revealed a surprising trend: in many cases, a well-prompted single-agent CoT baseline can outperform or perform on par with these complex, multi-agent architectures (Wang et al., 2024; Yao & Yadav, 2025). We also show these observations in our results (Table (1)) This finding is significant, as CoT is one of the foundational techniques for LLM reasoning. Its continued competitiveness suggests that the core reasoning unit—the individual thought or step—is of paramount importance. Arguably, the majority of recent research efforts have been dedicated to designing more elaborate MAS frameworks, while the fundamental CoT baseline has remained largely unchanged (Creswell et al., 2022; Chen et al., 2024). Our work pivots from this trend to focus on fundamentally reshaping and enhancing the CoT paradigm for the agentic era by redefining the nature of each reasoning step.

In this work, we introduce the Agentic Reasoning Module (ARM), a novel sequential reasoning approach where each granular step is executed by a specialized, self-contained reasoning agent. The core motivation is to elevate the "thinking" steps of CoT from simple textual continuation to the execution of a sophisticated, agentic block. This block is not manually designed but is instead automatically discovered through an evolutionary process. Starting with a basic CoT procedure, the module is iteratively mutated and refined based on its performance on a generic validation dataset of reasoning problems, resulting in a robust and versatile reasoning procedure that can be applied recursively at each step of solving a challenging multi-step problem.

The prevailing paradigm for MAS design often leads to systems that are highly domain-specific, with individual agents meticulously tuned for particular skills or tasks (Hu et al., 2025; Zhang et al., 2025b). While single-agent systems are generally considered more versatile, they too are often optimized for a narrow set of domains (LaMDAgen, 2025; ScribeAgent, 2024). In contrast, our work focuses on enhancing the universally applicable CoT framework. The agentic block within ARM can be optimized on any generic domain, yielding a general-purpose reasoning technique analogous to the original CoT. We demonstrate that this approach not only achieves superior performance but also exhibits greater generalizability. As we show, MAS built with ARM significantly outperform prominent MAS approaches across diverse agentic datasets without domain-specific tuning.

Our methodology uses the simple yet powerful CoT as a starting seed for the evolutionary discovery of ARM. A meta-agent orchestrates this process, performing a tree search over the code space of possible reasoning modules. Mutations and evolutions are guided by a reflection mechanism that analyzes execution traces from previous attempts, identifying weaknesses and proposing targeted improvements. Furthermore, this meta-agent discovers global strategies to orchestrate collaborations between parallel ARM reasoning traces, effectively creating a high-performance MAS from optimized, homogeneous building blocks. Overall, our work underscores the immense potential of evolving fundamental reasoning methodologies like CoT, presenting a more robust and scalable alternative to the development of increasingly complex and fragile heterogeneous MAS systems. Key contributions of our work are as follows:

- We present the Agentic Reasoning Module (ARM), an evolved and enhanced version of Chain-of-Thought reasoning. We demonstrate that systems built with ARM substantially outperform existing manually designed and automatically discovered multi-agent systems on complex reasoning tasks.
- We show that ARM is a significantly more generalizable reasoning module. MAS constructed with ARM maintain high performance across different underlying foundation models and task domains without requiring re-optimization, highlighting its robustness.
- We provide a rigorous justification and detailed ablations on the validity of our training objective demonstrating the effectiveness of the proposed MAS discovery strategy.

## 2 RELATED WORKS

**Single-Agent and Multi-Agent Reasoning Systems** The landscape of LLM-based reasoning is broadly divided into single-agent and multi-agent paradigms. Single-agent systems have demonstrated remarkable capabilities by augmenting the core LLM with sophisticated reasoning and action frameworks. A prominent example is the ReAct framework, which interleaves reasoning steps with actions, enabling the agent to interact with external tools like search engines to gather information and refine its reasoning process (Yao et al., 2023b). Other approaches have focused on enhancing single agents with self-reflection and memory to learn from past mistakes and improve performance iteratively (Shinn et al., 2023; Madaan et al., 2023). While these systems are powerful, their development has often focused on narrower tasks, such as tool-based search, retrieval, and question answering, rather than general-purpose complex reasoning.

In parallel, Multi-Agent Systems (MAS) have emerged as a dominant approach for tackling highly complex problems, often outperforming single-agent counterparts (Park et al., 2023; Qian et al., 2023). Frameworks like AutoGen (Wu et al., 2023), Camel (Li et al., 2023), and MetaGPT (Hong et al., 2023) orchestrate multiple LLM-powered agents, each assigned a specialized role (e.g., programmer, critic, tester). These agents collaborate, debate, and synthesize information to produce solutions for tasks like software development and complex reasoning. A key characteristic of these systems is their heterogeneous nature; each agent is distinct, with a manually engineered role and persona, connected through a predefined and often complex communication topology. In stark contrast, our ARM-based approach constructs a powerful MAS from homogeneous building blocks. The ARM itself is a self-contained, versatile reasoning module that is applied repeatedly, acting as the fundamental unit of thought for all "agents" in the system, thereby simplifying the design while enhancing generalizability.

**The Surprising Efficacy of Simple Reasoning Baselines** Despite the architectural complexity of many state-of-the-art MAS, a critical and recurring observation is the surprising competitiveness of simple reasoning baselines (Dubey et al., 2023). Foundational techniques like Chain-of-Thought (CoT) (Wei et al., 2022), and simple extensions like Self-Consistency (CoT-SC) which samples multiple reasoning chains and takes a majority vote (Wang et al., 2022), often achieve performance on par with, or even superior to, intricate multi-agent frameworks (Zhang et al., 2025a). This phenomenon is particularly pronounced with the advent of increasingly powerful frontier foundation models (Ke et al., 2025). As these models develop stronger native reasoning abilities, the high-level conceptual guidance provided by a simple CoT prompt is often sufficient to unlock their full potential, rendering the overhead of complex agent orchestration less impactful. This suggests that the primary bottleneck is not necessarily the high-level orchestration strategy but the quality and robustness of the fundamental, step-by-step reasoning process. Our work is directly motivated by this insight, positing that evolving the core reasoning operator—the "thought" in the chain—is a more fruitful direction than designing ever-more-complex superstructures around a static, simple CoT unit.

**Automated Design of Multi-Agent Systems** Recognizing the significant manual effort required to design effective MAS, recent research has explored automating this process. Approaches like ADAS (Hu et al., 2025), Aflow Zhang et al. (2025b), and Flow-Reasoner (Gao et al., 2025) aim to automatically discover the optimal agent roles and their interaction topology for a given task domain. However, these techniques suffer from two major drawbacks. First, they are computationally expensive, requiring a costly re-discovery process for each new task domain. Second, the discovered systems are often highly specialized and brittle, tuned specifically for the validation data of a single domain. As our results will later show, with the latest generation of foundation models, these automatically discovered systems can be outperformed by simple CoT baselines. Our work diverges from this paradigm. Instead of discovering a complex, domain-specific agent topology, we focus on discovering a single, domain-agnostic reasoning module (ARM). This ARM acts as a universal, high-quality building block that provides superior performance and generalizability without the need for task-specific rediscovery, offering a more scalable and robust path forward for MAS design.

**LLM based Prompt Optimizers** Recent research has focused on LLMs as prompt optimizers, leveraging their generative and reasoning capabilities to automatically improve prompts within a fixed workflow Zhou et al. (2023); Yang et al. (2024); Khattab et al. (2024); Guo et al. (2024); Novikov et al. (2025); Fernando et al. (2024). Notably, evolutionary approaches coupled with deep reflection over rollouts, such as in GEPA Agrawal et al. (2025), have been shown to offer signifi-

cant advantages in sample efficiency compared to methods that involve updating model weights via Reinforcement Learning.

### 3 METHODOLOGY: DISCOVERING THE AGENTIC REASONING MODULE

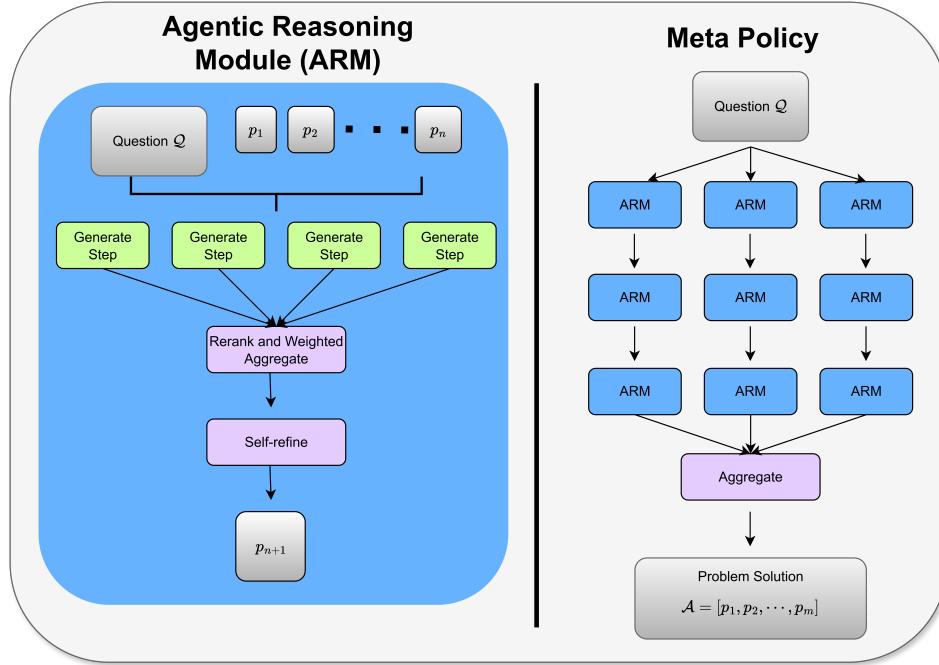


Figure 1: An illustration of the proposed ARM module on the left and the meta policy on the right using "Self refine" as an example MAS. The ARM module takes a question and previous reasoning steps and executes a MAS to get the next step. The meta policy uses ARM as a sub-module and orchestrates the 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).

We introduce the **Agentic Reasoning Module (ARM)**, a self-contained, code-based multi agentic system designed to execute a single, granular step within a complex reasoning process. ARM is conceived as a structured, agentic replacement for a single step in a Chain of Thought (CoT) sequence Wei et al. (2022). While standard CoT prompts an LLM to generate the next reasoning step via naive, monolithic textual generation, an ARM employs an internal multi-agent system (MAS) to produce reasoning steps with greater structure and control.

Following prior work, Hao et al. (2023); Zhang et al. (2024), we define the multi-agentic system as a programming module - a self contained Python function block, while allowing for structured control flow and access to essential APIs such as calling an external LLM, structuring the role and the prompt, and input/output format expectations. Functionally, an ARM accepts the initial problem statement and prior reasoning steps as input, and continues the reasoning until the next logical step in the solution.

#### 3.1 A DECOMPOSABLE FRAMEWORK FOR AGENTIC REASONING

Let the distribution over problem-solution pairs be  $\mathcal{D}$  over  $(\mathcal{Q}, \mathcal{A})$ . A solution  $\mathcal{A}$  consists of a sequence of reasoning steps  $[p_1, p_2, \dots, p_N]$ , where each step  $p_i$  belongs to the space of all possible reasoning steps  $\mathcal{P}$ . We model the problem-solving process with two key functions:

\* **The Step-Generator Module ( $m \in \mathcal{M}$ ):** This is a program that performs a single step of reasoning. It takes the problem question  $q \in \mathcal{Q}$  and the history of previous reasoning steps  $p_{in} \in \mathcal{P}^*$  as input and returns the next reasoning step  $p_{out} \in \mathcal{P}$ . Its signature is  $m : \mathcal{Q} \times \mathcal{P}^* \rightarrow \mathcal{P}$ . An **Agentic Reasoning Module (ARM)** is a structured, code-based implementation of such a module, which can itself be a self-contained MAS.

\* **The Meta-Policy** ( $\pi \in \Pi$ ): This is a higher-order program that defines the overarching strategy. It takes a question  $q$  and a specific step-generator module  $m$  and orchestrates calls to  $m$  to generate a complete solution  $a \in \mathcal{A}$ . Its signature is  $\pi : \mathcal{Q} \times \mathcal{M} \rightarrow \mathcal{A}$ .

Within this framework, standard Chain of Thought (CoT) can be seen as a simple baseline pairing. It uses a basic step-generator,  $m_{CoT}$ , which is a single call to an LLM, and a simple **recursive meta-policy**,  $\pi_{Rec}$ , which applies  $m_{CoT}$  repeatedly until a final answer is produced. Our approach independently discovers a more powerful module  $m^*$  (the ARM) and a more sophisticated meta-policy  $\pi^*$ .

### 3.2 DISCOVERING THE OPTIMAL STEP-GENERATOR ( $m^*$ )

Our primary goal is to find a step-generator module  $m^*$  that is a general-purpose and superior replacement for the simple text generation step in  $m_{CoT}$ .

We can formalize a single reasoning step as an update function,  $U_{m,q}$ , that appends the output of module  $m$  to the current reasoning history  $h$ :

$$U_{m,q}(h) = h \cdot [m(q, h)]$$

where  $\cdot$  denotes list concatenation. A full,  $n$ -step reasoning trace generated by the recursive policy  $\pi_{Rec}$  is thus the  $n$ -fold composition of this update function:  $\pi_{Rec}(q, m) = U_{m,q}^n(\emptyset)$ .

Ideally, we would discover the optimal module  $m^*$  by maximizing the expected reward  $\mathcal{R}$  over the entire problem-solving trace:

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

However, optimizing this objective directly is intractable due to two main challenges: 1. **Difficult Credit Assignment**: The reward is observed only at the end of a long sequence of steps, making it difficult to determine which specific application of  $m$  was responsible for the final outcome. 2. **Unconstrained Search Space**: The space of possible code-based modules  $\mathcal{M}$  is vast, making an unguided search highly inefficient.

To address this, we introduce a practical **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 generated by the baseline  $m_{CoT}$ . Specifically, we replace a small, contiguous block of  $l$  steps within an  $n$ -step CoT trace with our candidate module  $m$ . The optimization problem becomes:

$$m^* = \operatorname{argmax}_{m \in \mathcal{M}} \mathbb{E}_{(q,a) \sim \mathcal{D}} [\mathcal{R}(U_{m_{CoT},q}^{n-l-i} \circ U_{m,q}^l \circ U_{m_{CoT},q}^i(\emptyset), a)]$$

where  $n = |\pi_{Rec}(q, m_{CoT})|$  is the length of the reference CoT trace, and the starting index  $i$  is chosen randomly from  $[0, n - 1]$ . This formulation isolates the performance contribution of  $m$  to a small window, enabling direct credit assignment. Furthermore, the surrounding CoT context provides a powerful inductive bias, constraining the search to modules that behave as effective, incremental reasoning steps. This mirrors the conservative policy-improvement principle Kakade & Langford (2002), where a candidate policy is evaluated under a stable reference distribution to guarantee monotonic improvement. Likewise, the scaffold constrains module updates within a fixed Chain-of-Thought context, ensuring stable, incremental reasoning gains. In our experiments, we find  $l = 3$  works well, as it is long enough to expose the module  $m$  to critical compositional patterns— $(U_{m_{CoT},q} \circ U_{m,q})$ ,  $(U_{m,q} \circ U_{m,q})$ , and  $(U_{m,q} \circ U_{m_{CoT},q})$ —while keeping the optimization tractable.

### 3.3 DISCOVERING THE OPTIMAL META-POLICY ( $\pi^*$ )

While an optimized step-generator  $m^*$  improves the quality of each reasoning step, the high-level strategy  $\pi$  that orchestrates these steps is equally critical. A simple recursive policy,  $\pi_{Rec}$ , may be suboptimal for complex problems that could benefit from strategies like parallel rollouts (for self-consistency) or iterative refinement loops Wang et al. (2023); Madaan et al. (2023).

Searching for an optimal meta-policy  $\pi^*$  by repeatedly evaluating candidates with the full, complex  $m^*$  module is computationally prohibitive. Therefore, we adopt a surrogate-based approach here as well. We search for the optimal meta-policy  $\pi^*$  using the fast and computationally cheap baseline step-generator,  $m_{CoT}$ , as a stand-in for  $m^*$ .

This zero-shot transfer from  $m_{CoT}$  to  $m^*$  is effective because our step-generator optimization process (Section 3.2) 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 orchestrating the more powerful, but functionally analogous, steps of  $m^*$ . This allows us to efficiently explore the space of strategies, discovering sophisticated control flows like branching for parallel thought generation or conditional loops for verification, without incurring the high computational cost of using  $m^*$ .

### 3.4 REFLECTION-GUIDED EVOLUTIONARY SEARCH

We discover both the optimal step-generator  $m^*$  and meta-policy  $\pi^*$  using a unified **Reflection-Guided Evolutionary Search** algorithm. This algorithm performs a tree search over the programmatic space of valid Python modules, where each node in the tree represents a specific program. The search begins with a root node representing the baseline program ( $m_{CoT}$  for the step-generator search and  $\pi_{Rec}$  for the meta-policy search). The search then iteratively performs three steps:

1. **Selection:** A parent node (program)  $p_{parent}$  is sampled from the current tree  $\mathcal{T}$  using temperature sampling based on its validation performance.
2. **Expansion:** A new child program is generated by a **Reviewer Agent**, an LLM-based agent that reflects on the parent program's execution traces, correctness, and mutation history to propose a targeted code modification.
3. **Evaluation:** The newly generated program is evaluated to obtain its average reward  $\bar{\mathcal{R}}$ . For a step-generator module, we use the scaffolded objective from Section 3.2. For a meta-policy, we evaluate its performance on a full problem rollout using  $m_{CoT}$  as the step-generator.

This entire process is summarized in Algorithm 1.

#### 3.4.1 THE REVIEWER AGENT

The expansion step is driven by a two-stage **Reviewer Agent** that intelligently mutates existing programs. This agent consists of two LLM-based components:

**Critic:** The Critic analyzes execution traces from the parent program. It identifies logical errors, inefficiencies, or patterns of failure, providing a concise, natural-language analysis of the program's strengths and weaknesses.

**Designer:** The Designer acts as the mutation operator. It takes the original program's code, its performance history, and the Critic's analysis as input. Based on this information, it proposes a single, targeted code modification aimed at addressing the identified issues, generating a complete, syntactically valid Python class for the new program.

This reflection-driven process ensures that the search evolves programs purposefully, rather than through random mutations, leading to more efficient discovery of high-performance modules and policies. The prompts used for the Critic and Designer are detailed in the Appendix.

## 4 ARM SEARCH ALGORITHM

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

## 5 EXPERIMENTS

### 5.1 BENCHMARKS

We evaluated our baselines and approach on multiple complex reasoning datasets. To assess complex mathematical reasoning capabilities, we utilized widely studied *American Invitational Mathematics Examination (AIME)*<sup>1</sup> and the *Harvard-MIT Mathematics Tournament (HMMT)*<sup>2</sup> datasets.

<sup>1</sup>[https://huggingface.co/datasets/MathArena/aime\\_2025](https://huggingface.co/datasets/MathArena/aime_2025)

<sup>2</sup>[https://huggingface.co/datasets/MathArena/hmmt\\_feb\\_2025](https://huggingface.co/datasets/MathArena/hmmt_feb_2025)

---

**Algorithm 1** Reflection-Guided Search

---

```

1: Input: Initial program  $p_{root}$  (e.g.,  $m_{CoT}$  or  $\pi_{Rec}$ ), evaluation function EVALUATE( $\cdot$ ), total iterations  $K$ , exploration constant  $C$ .
2: Initialize:
3: Tree  $\mathcal{T}$  with a single node for  $p_{root}$ .
4:  $p_{root}.\bar{\mathcal{R}} \leftarrow \text{EVALUATE}(p_{root})$   $\triangleright$  Evaluate the baseline program on a validation batch
5:  $p_{root}.N \leftarrow 1$   $\triangleright$  Initialize visit count for the root
6: for  $t = 1$  to  $K$  do
7:    $\triangleright 1. Select a parent program to mutate$ 
8:    $P(p_i) \leftarrow \frac{\exp(p_i.\bar{\mathcal{R}}/T)}{\sum_{j \in \mathcal{T}} \exp(p_j.\bar{\mathcal{R}}/T)}$ 
9:    $p_{parent} \leftarrow \text{Sample}(\mathcal{T}, P)$ 
10:   $\triangleright 2. Expand the tree via reflection$ 
11:  traces  $\leftarrow \text{EXECUTE}(p_{parent})$   $\triangleright$  Collect execution traces
12:  history  $\leftarrow \text{GETMUTATIONHISTORY}(p_{parent})$ 
13:   $p_{new} \leftarrow \text{REVIEWERAGENT}(p_{parent}, \text{traces}, \text{history})$ 
14:   $\triangleright 3. Evaluate the new program$ 
15:   $p_{new}.\bar{\mathcal{R}} \leftarrow \text{EVALUATE}(p_{new})$ 
16:   $p_{new}.N \leftarrow 1$ 
17:   $\triangleright 4. Update tree and statistics$ 
18:   $\mathcal{T}.\text{ADDCHILD}(p_{parent}, p_{new})$ 
19:   $p_{parent}.N \leftarrow p_{parent}.N + 1$ 
20: end for
21:
22: return  $\underset{p_i \in \mathcal{T}}{\text{argmax}} (p_i.\bar{\mathcal{R}})$   $\triangleright$  Return the program with the highest empirical reward

```

---

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 and robustness against data contamination, we used *LiveBench Reasoning*<sup>3</sup>, a dynamic benchmark with continuously evolving questions (Jain et al., 2024).

## 5.2 BASELINES

We compare our methodology against two distinct groups of multi-agent systems (MAS) baselines: popular handcrafted MAS systems and leading automated MAS generation approaches.

### 5.2.1 HANDCRAFTED MULTI-AGENT SYSTEMS:

We compare against several strong reasoning baselines. **Chain of Thought (CoT)** Wei et al. (2022) serves as the fundamental baseline, solving tasks through iterative textual reasoning. **CoT-Self Consistency (CoT-SC)** Wang et al. (2023) improves upon CoT by generating  $n = 12$  parallel reasoning rollouts and selecting the final answer via a majority vote. **Self-Refine** Madaan et al. (2023) employs a feedback loop where a Large Language Model (LLM) iteratively critiques and refines its own output. Lastly, **LLM-Debate** Du et al. (2023) initializes multiple LLM agents with diverse roles to generate different reasoning paths, fostering a debate to converge on a final solution.

### 5.2.2 AUTOMATED MULTI-AGENT SYSTEMS:

These baselines include the two leading code based MAS generation approaches: **ADAS** Hu et al. (2025) and **AFlow** Zhang et al. (2025b). These methods employ search algorithms to automatically discover the optimal agent roles and their complex interaction topology for a given task domain

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

<sup>3</sup><https://huggingface.co/datasets/livebench/reasoning>

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 in Table 1.

### 5.3 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. During validation and inference, we 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).

### 5.4 TRAINING

Our training process is designed to independently discover the two core components of our framework: the optimal step-generator module ( $m^*$ ) and the optimal meta-policy ( $\pi^*$ ). This decoupled approach allows us to first forge a powerful, general-purpose reasoning module and then learn a sophisticated strategy to orchestrate it, all without requiring expensive, domain-specific annotations.

**Validation Dataset:** For both discovery processes, we utilize the a subset (1000 samples) of the Math and Science splits of the Open-or-Mixture-of-Thoughts HuggingFace (2025) dataset, a general-purpose instruction-following dataset. Our method requires only a one-time, domain-agnostic training phase. The same resulting code artifacts are then deployed across all benchmark domains and foundation models without any task-specific fine-tuning or re-optimization, underscoring the robustness and versatility of our method.

**Step-Generator ( $m^*$ ) Discovery:** We discover the ARM module by employing the Reflection-Guided Evolutionary Search detailed in Algorithm 1. The search is initialized with a basic Chain-of-Thought module ( $m_{CoT}$ ) and iteratively evolves it by maximizing the scaffolded surrogate objective from Section 3.2. This objective evaluates candidate modules within the context of a baseline CoT trace, enabling efficient and stable optimization.

**Meta-Policy ( $\pi^*$ ) Discovery:** The meta-policy is discovered independently using the same evolutionary search algorithm. To ensure computational tractability, this search is performed using the simple and fast baseline module,  $m_{CoT}$ , as a surrogate for the more complex  $m^*$  (as justified in Section 3.3). This allows us to efficiently explore the space of high-level strategies and discover a sophisticated meta-policy that can be seamlessly paired with the optimized ARM module.

## 6 RESULTS

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

- (1) **Naive Operators outperform MAS:** Simple basic operators such as CoT, Self-refine, LLM-Debate outperform complex MAS systems like AFlow and ADAS. This highlights an important concern regarding the practicality of recent advancements in MAS. On the other hand, simple reasoning operators such as CoT perform substantially better across tasks, and varied families of LLMs. Our ARM based reasoning approach is step forward to revitalize traditional yet strong reasoning methods like CoT, by advancing their reasoning steps with agentic blocks. Our ARM based approach further improves up the CoT performance and achieves best results all the datasets.
- (2) **ARM achieving top performance:** ARM consistently outperforms all of the operator baselines. Specifically, in complex datasets such as AIME and HMMT, ARM consistently outperforms existing MAS approaches and all the existing baseline operators. This emphasizes the benefits and strong potential of revitalizing proven traditional reasoning methods like CoT.
- (3) **Effects from stronger foundation LLM:** We first note an important observation that with stronger LLMs such as GPT-4o, simple operators such as CoT and CoT-SC outperform complex MASes. Our ARM based reasoning approach further pushes the best performance over the baseline operators with both recent stronger frontier models such as GPT4.1-nano / GPT-4o and older benchmark models such as LLaMa-3.3-70B.

Model	Method	MATH-500	AIME2S	HMMT2S	GPQA	LiveBench	Average
GPT-4.1-nano	CoT	82.0%	15.1%	9.9%	50.0%	33.1%	38.0%
	CoT-SC	<b>86.2%</b>	<u>21.9%</u>	13.5%	50.6%	36.9%	41.8%
	Self-Refine	84.2%	17.2%	9.4%	50.0%	28.1%	37.8%
	LLM-Debate	84.2%	15.1%	<u>16.7%</u>	52.5%	33.8%	40.5%
	ADAS (test set)	79.8%	12.0%	5.2%	48.1%	31.2%	35.3%
	ADAS (1000-sample)	<u>77.3%</u>	0.0%	<u>6.8%</u>	46.8%	29.4%	32.0%
	AFlow (test set)	74.5%	18.8%	12.0%	39.9%	30.6%	35.2%
	AFlow (1000-sample)	77.0%	16.7%	10.4%	51.3%	30.6%	37.2%
	<b>ARM (Ours)</b>	82.0%	18.2%	14.6%	<u>60.1%</u>	<u>39.4%</u>	<u>42.9%</u>
	<b>ARM + MP (Ours)</b>	<u>86.0%</u>	<b>23.4%</b>	<b>22.4%</b>	<b>61.4%</b>	<b>45.6%</b>	<b>47.8%</b>
GPT-4.0	CoT	75.0%	7.3%	0.5%	53.8%	46.2%	36.6%
	CoT-SC	<u>81.8%</u>	12.5%	2.1%	<u>53.2%</u>	42.5%	38.4%
	Self-Refine	77.2	6.8%	2.6%	53.8%	37.5%	35.6%
	LLM-Debate	<u>81.8%</u>	9.9%	3.1%	56.3%	<u>47.5%</u>	39.7%
	ADAS (test set)	65.5%	1.0%	0.0%	46.2%	38.8%	30.3%
	ADAS (1000-sample)	69.0%	0.0%	0.5%	46.8%	41.9%	31.6%
	AFlow (test set)	75.5%	9.9%	3.6%	53.8%	41.9%	36.9%
	AFlow (1000-sample)	48.8%	9.4%	0.0%	50.6%	45.0%	30.8%
	<b>ARM (Ours)</b>	78.3%	<u>13.5%</u>	<u>5.7%</u>	<u>59.5%</u>	<u>47.5%</u>	<u>40.9%</u>
	<b>ARM + MP (Ours)</b>	<b>82.0%</b>	<u>17.2%</u>	<b>9.4%</b>	<b>60.1%</b>	<b>51.9%</b>	<b>44.1%</b>
LLAMA-3.3-70B	CoT	75.0%	6.8%	3.1%	50.0%	38.1%	34.6%
	CoT-SC	78.5%	4.2%	<u>5.7%</u>	<b>53.2%</b>	45.0%	37.3%
	Self-Refine	77.8%	6.8%	4.2%	<u>51.3%</u>	<u>46.9%</u>	37.4%
	LLM-Debate	79.0%	5.7%	4.2%	50.6%	46.2%	37.1%
	ADAS (test set)	67.2%	3.1%	0.0%	47.5%	37.5%	31.0%
	ADAS (1000-sample)	22.2%	3.1%	0.5%	42.4%	46.2%	22.9%
	AFlow (test set)	65.2%	4.7%	0.0%	46.8%	38.1%	31.0%
	AFlow (1000-sample)	63.2%	7.2%	3.1%	46.8%	15.6%	27.2%
	<b>ARM (Ours)</b>	<u>80.0%</u>	<b>8.3%</b>	<u>5.2%</u>	49.6%	46.2%	<u>37.9%</u>
	<b>ARM + MP (Ours)</b>	<b>80.8%</b>	<u>7.8%</u>	<b>6.8%</b>	50.0%	<b>50.0%</b>	<b>39.1%</b>

Table 1: Main results on four complex 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.

## 7 ANALYSES

To understand the sources of ARM’s effectiveness, we performed two key analyses. First, we provide empirical evidence that our search objective discovers fundamentally more reliable reasoning modules by minimizing their per-step error rate. Secondly, we show the validity of our efficient, decoupled training strategy by demonstrating that the learned meta-policy transfers zero-shot from a simple surrogate to the final ARM, yielding significant performance gains.

### 7.1 EMPIRICAL VALIDATION OF THE STEP-GENERATOR OBJECTIVE

To empirically validate our theoretical claim (Appendix A) that the scaffolded objective minimizes per-step error, we conducted a targeted ablation study. We executed the top five discovered step-generator modules for a single step, starting from *critical reasoning junctures* identified by an LLM-judge (GPT-OSS-20B) within baseline  $m_{CoT}$  traces. The error rate of each single-step output was then evaluated. As shown in Figure 1, a module’s rank, determined by our objective, strongly correlates with a lower per-step error rate at these critical points. This result confirms that our search process successfully discovers modules that are fundamentally more robust at a granular level, validating the core mechanism behind ARM’s performance.

### 7.2 EMPIRICAL VALIDATION OF META-POLICY TRANSFER

Our methodology relies on a crucial transfer: a meta-policy trained with the simple  $m_{CoT}$  module is deployed zero-shot with the powerful, discovered  $m^*$  module. The theoretical justification in Appendix B posits this transfer is effective due to two factors: (1) the inherent superiority of the  $m^*$  module, and (2) its ability to guide the reasoning process into more productive states. We designed an experiment to empirically disentangle and verify these two sources of gain.

To do this, we measure and compare three distinct performance configurations. First, we establish a **baseline performance** using the meta-policy with the simple  $m_{CoT}$  module. Second, to isolate the **pure module improvement gain**, we measure the performance of the powerful  $m^*$  module when it takes over from intermediate reasoning states generated by the baseline  $m_{CoT}$ . Finally, we measure the **full system performance** of the meta-policy paired with  $m^*$  from the start.

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 table compares performance using the simple surrogate  $m_{CoT}$  (**CoT Baseline**) versus the powerful ARM module  $m^*$  (**Meta Policy**). The intermediate **CoT→Meta** column isolates the performance gain from the superior  $m^*$  module alone by evaluating it on states generated by the baseline.

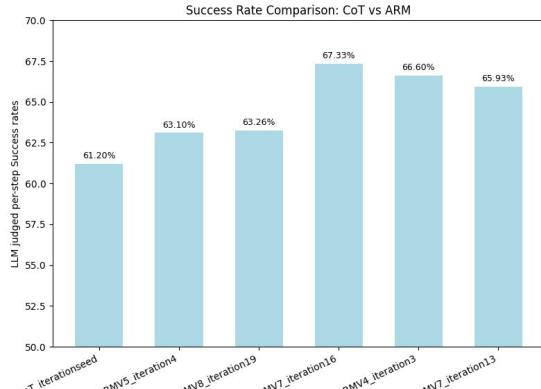


Figure 3: Comparison of LLM judged per-step success rates between the baseline *Chain-of-Thought* (CoT) and multiple *ARM* (CriticChainOfThought) variants. CoT appears first, followed by ARM variants ordered by final performance.

The results, shown in Figure 2, confirm our hypothesis with a clear performance hierarchy. The baseline system performs worst, followed by a significant improvement from simply swapping to the  $m^*$  module. The best performance is achieved by the full system, which benefits from both the better module and its ability to find a better reasoning path. This empirically validates the two conditions for successful transfer outlined in Appendix A and confirms the effectiveness of our decoupled discovery strategy.

## 8 CONCLUSION

We introduced ARM, a modular agentic reasoning framework that revitalizes the traditional Chain-of-Thought (CoT) paradigm by augmenting it with lightweight agentic blocks. Through extensive experiments, we demonstrated that simple operators such as CoT and Self-Refine not only remain highly competitive but, in many cases, outperform complex Multi-Agent Systems (MAS), highlighting the growing gap between empirical performance and the perceived promise of increasingly elaborate MAS designs. Our results show that ARM consistently advances the performance of CoT across diverse reasoning tasks and model families, establishing top-performing results.

Beyond empirical improvements, ARM sheds light on an important perspective: improving the granular step by step reasoning process holds the key to progress in reasoning systems. 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.
- Baian Chen, Chang Li, Zhuo Li, Jianing Wang, Yafen Tian, Rui Wang, and Xin Wang. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2403.01925*, 2024.
- Weize Chen, Yusheng Zhang, Zihan Zhang, Cheng Liu, Zipeng Zheng, Chen Qian, Yufan Zhao, Yufan Cong, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. *arXiv preprint arXiv:2308.10848*, 2023.
- Antonia Creswell, Murray Shanahan, and Irina Higgins. Selection-inference: Exploiting large language models for interpretable logical reasoning. *arXiv preprint arXiv:2205.09712*, 2022.
- Yihong Dong, Xue Wang, Ge Jiang, Zhiping Liu, Cilin Zhang, Peiyu Wang, and Yi Zhang. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.
- 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>.
- Rishabh Dubey, Daochen Zha, Lingjiao Wu, and Aditya Grover. Revisiting the gold standard: A critical look at multi-agent systems for discovery. *arXiv preprint arXiv:2310.10653*, 2023.
- 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>.
- Hongcheng Gao, Yue Liu, Yufei He, Longxu Dou, Chao Du, Zhijie Deng, Bryan Hooi, Min Lin, and Tianyu Pang. Flowreasoner: Reinforcing query-level meta-agents. *arXiv preprint arXiv:2504.15257*, 2025.
- Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. In *International Conference on Learning Representations (ICLR)*, May 2024.
- Shibo Hao, Yi Gu, Haodi Ma, Joshua Wang, Zhen Chen, and Zhaofeng Wang. Reasoning with language model is planning with world model. *arXiv preprint arXiv:2305.14992*, 2023.
- Sirui Hong, Xiawu Zheng, Jonathan Chen, Kechen Yang, Yida Li, Weya Su, Chen Wang, Ceyao He, 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.

Zixuan Ke, Fangkai Jiao, Yifei Ming, Xuan-Phi Nguyen, Austin Xu, Do Xuan Long, Minzhi Li, Chengwei Qin, Peifeng Wang, Silvio Savarese, et al. A survey of frontiers in llm reasoning: Inference scaling, learning to reason, and agentic systems. *arXiv preprint arXiv:2504.09037*, 2025.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhaman, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2024.

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.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

LaMDAgent. Lamdagent: An autonomous framework for post-training pipeline construction, 2025.  
URL <https://arxiv.org/html/2505.21963v1>.

Guohao Li, Hasan Momin, Hasan Ground, ‘ Kian, et al. Camel: Communicative agents for" mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Kevin Hall, Luyu Gao, Sarah Wiegreffe, Uri Alon, Pengcheng Cai, 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/](https://www.llama.com/docs/model-cards-and-prompt-formats/llama3_3/), December 2024. Accessed: 2025-09-27.

Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Noah Chapman, George Dugan, Miljan Tworkowski, Croitoru Alfredo, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

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 Mädry, 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 Tootoochian, 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 Guaraci, 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 Singh, 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 Cunningham, 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 Wang, Yufan Cong, Cheng Liu, Weize Yu, Zipeng Zheng, Zihan Chen, Yapen Gu, 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.

John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In Francis Bach and David Blei (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 1889–1897, Lille, France, 07–09 Jul 2015. PMLR. URL <https://proceedings.mlr.press/v37/schulman15.html>.

ScribeAgent. Scribeagent: Fine-tuning open-source llms for enhanced web navigation, 2024. URL <https://blog.ml.cmu.edu/2024/12/06/scribeagent-fine-tuning-open-source-llms-for-enhanced-web-navigation/>.

Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 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.

Ruida Wang, Rui Pan, Yuxin Li, Jipeng Zhang, Yizhen Jia, Shizhe Diao, Renjie Pi, Junjie Hu, and Tong Zhang. Ma-lot: Multi-agent lean-based long chain-of-thought reasoning enhances formal theorem proving. *arXiv preprint arXiv:2503.03205*, 2025.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

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, Shaokun Li, Erkang Zhu, Beibin Li, Li Jiang, et al. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations (ICLR)*, May 2024.

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.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Sha, Silvio Savarese, and Sima an. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Sha, Narasimhan Karthik, and Sima an. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, Lei Bai, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, 2025a. Oral Presentation (Top ~1%).

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)*, 2025b. Oral Presentation (Top 1.8%).

Lei Zhang, Feng Xu, Zongyi Yu, Chen Zhu, and Yu Qian. Agent-flow: A flexible and scalable multi-agent platform for real-life tasks. *arXiv preprint arXiv:2402.17779*, 2024.

Wei Zhang, Chen Liu, Ananya Patel, Ming Zhao, and Jie Huang. Flowreasoner: Automatic multi-agent system generation for complex reasoning. *arXiv preprint arXiv:2502.08123*, 2025c. Accessed: YYYY-MM-DD.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *International Conference on Learning Representations (ICLR)*, November 2023.

## A 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. 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 analyze an idealized formulation of the problem as a Markov Decision Process (MDP). This abstracts away the underlying complexities of the text generation and focuses on the dynamics of per-step error and state distribution shift. This analysis provides a formal argument for the soundness of our proposed search objective.

### A.1 AN IDEALIZED MDP MODEL OF STEP-WISE REASONING

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

- **State Space ( $S$ ):** The state space  $S = S_{ok} \cup S_{fail} \cup S_{done}$  is partitioned into three disjoint subsets:
  - $S_{ok}$ : A state  $s \in S_{ok}$  represents a partial reasoning trace  $q, p_1, \dots, p_k$  that is on a valid path to a solution.
  - $S_{fail}$ : A state  $s \in S_{fail}$  has made a critical reasoning error from which recovery is not possible (including terminal states where the reasoning chain ended up on the wrong answer). This is an absorbing region.
  - $S_{done}$ : A state  $s \in S_{done}$  represents a reasoning path that has successfully ended on the right answer. This is an absorbing region.
- **Action Space ( $A$ ):** For a fixed meta-policy  $\pi_{Rec}$  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  $s \in S_{ok}$ : i.e., invokes a step-generator module  $m$  to produce the next reasoning step. Thus, the action space is a singleton  $\mathcal{A} = \{\text{generate\_step}\}$ . Hence, the choice of the module  $m$  fully defines the transition dynamics of the MDP.
- **Reward Function ( $R$ ):** A sparse reward function with  $R = 1$  given upon transition to a done state.
  - $R(s) = 1$  if  $s \in S_{done}$  and 0 otherwise.
- **Transition Dynamics ( $P$ ):** update function (see 3.2.1)  $U_m$ , where  $m \in \mathcal{M}$  is characterized by its state-dependent error rate,  $\epsilon_m(s)$ , which is the probability of making a catastrophic irrecoverable error from state  $s$ . Additionally, we introduce For any state  $s \in S$ , the transition probabilities to the next state  $s'$  are defined as:

$$- P(s'|s, m) = \begin{cases} p_{done}(s) & \text{if } s' \in S_{done} \\ \epsilon_m(s) & \text{if } s' \in S_{fail} \\ 1 - p_{done}(s) - \epsilon_m(s) & \text{if } s' \in S_{ok} \end{cases}$$

Note that  $p_{done}(s)$  is the intrinsic probability of finishing the task by transitioning into a done state from state  $s$ . In practice, this is when the model emits //boxed[Correct Answer]. This can be assumed to be a property of the task and the progress so far, rather than the generator itself.

- **Value Function:** The value of a state  $s$  under module  $m$ , denoted by  $V_m(s)$  is the probability of eventually reaching a success state in  $S_{done}$  (thus receiving a reward) starting from  $s$ .

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

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

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

## A.2 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  $t \in \{i, \dots, i + \ell - 1\}$  while keeping  $m_{\text{CoT}}$  before and after:

$$\underbrace{U_{m_{\text{CoT}}}^* \circ (U_m^\ell) \circ U_{m_{\text{CoT}}}^i}_{\text{"baseline--candidate--baseline"}.}$$

Let  $d_{\text{CoT},t}$  be the state distribution at step  $t$  along the (unperturbed) baseline trace. Define the single-step advantage  $A_m(s) := V_m(s) - V_{\text{CoT}}(s)$ .

**Assumption 1** (Finite-horizon conditional stability). *Within the scaffold window and for states along the baseline trace ( $s \sim d_{\text{CoT},t}$ ), the conditional next-state distributions inside  $\mathcal{S}_{ok}$  remain close:*

$$\mathbb{E}_{s \sim d_{\text{CoT},t}} \left[ D_{\text{TV}}(T_m(\cdot | s, \mathcal{S}_{ok}) \| T_{\text{CoT}}(\cdot | s, \mathcal{S}_{ok})) \right] \leq \beta_{ok} \quad \forall t \in \{i, \dots, i + \ell - 1\}.$$

- **Remark:** This constraint requires that, conditional on the trajectory staying in  $\mathcal{S}_{ok}$ , the distribution over possible next states under module  $m$  is close on average (in total variation distance) to the distribution under  $m_{\text{CoT}}$ .

This assumption is grounded in the mechanics of the prompted LLMs. The scaffold provides a strong contextual prior (the preceding CoT steps) and acts as a powerful inductive bias, strongly constraining the module to generate a next step that is stylistically and logically consistent with the atomic reasoning style of CoT. For instance, by defining variables in the same format, using consistent LaTeX for equations, or following an established deductive pattern (e.g., ‘Let  $x$  be..., then it follows that  $y\dots$ , therefore  $z\dots$ ’ etc.). This incentivizes the LLM to generate a coherent and logically plausible continuation—a necessary condition for remaining in  $\mathcal{S}_{ok}$ . Therefore, any “successful” module  $m$  must, by necessity, learn to mimic the local *successful* behavior of  $m_{\text{CoT}}$  while minimizing the transition probability to hazard states  $\mathcal{S}_{fail}$ . This is empirically supported by the examples shown in Appendix-B.

**Proposition 1** (Scaffolded objective optimizes per-step error rate). *Let  $w_t$  be the probability the baseline remains in  $\mathcal{S}_{ok}$  up to step  $t$ . Then for a universal constant  $C > 0$ ,*

$$V_{\text{scaffold}}(m) - V_{\text{scaffold}}(m_{\text{CoT}}) \geq \sum_{t=i}^{i+\ell-1} \mathbb{E}_{s \sim d_{\text{CoT},t}} [A_m(s)] - C \sum_{t=i}^{i+\ell-1} w_t \beta_{ok}.$$

Moreover, under Assumption 1,  $A_m(s)$  is dominated by error-rate reduction:

$$A_m(s) \approx (\varepsilon_{\text{CoT}}(s) - \varepsilon_m(s)) \cdot \mathbb{E}_{s' \sim T_{\text{CoT}}(\cdot | s, \mathcal{S}_{ok})} [V_{\text{CoT}}(s')].$$

*Proof of Proposition 1.* By Assumption-1, the performance improvement within the  $\ell$ -step scaffold window is lower bounded by the cumulative advantage, minus a small penalty for the distribution shift on successful steps Kakade & Langford (2002); Schulman et al. (2015).

Side Note: For the purpose of a conservative lower bound, we treat this term as a penalty. However, in practice such conditional distributional shift may be beneficial: an advanced module  $m$ , potentially using mechanisms like self-consistency or debate, could guide the trajectory toward higher-value states even within  $\mathcal{S}_{ok}$ . This would add a positive contribution, but our main result does not rely on that stronger assumption.

For a step  $t$  and a baseline state  $s \in \mathcal{S}_{ok}$ , the Bellman equation with our transitions gives

$$V_m(s) = p_{\text{done}}(s) \cdot 1 + (1 - p_{\text{done}}(s) - \varepsilon_m(s)) \cdot \mathbb{E}_{s' \sim T_m(\cdot | s, \mathcal{S}_{ok})} [V_m(s')].$$

Subtract the corresponding identity for  $V_{\text{CoT}}(s)$  and rearrange:

$$\begin{aligned} A_m(s) &= V_m(s) - V_{\text{CoT}}(s) \\ &= \underbrace{(\varepsilon_{\text{CoT}}(s) - \varepsilon_m(s)) \mathbb{E}_{s' \sim T_{\text{CoT}}(\cdot | s, \mathcal{S}_{ok})} [V_{\text{CoT}}(s')]}_{\text{error-rate reduction term}} \\ &\quad + \underbrace{(1 - p_{\text{done}}(s) - \varepsilon_m(s)) (\mathbb{E}_{s' \sim T_m} [V_m(s')] - \mathbb{E}_{s' \sim T_{\text{CoT}}} [V_{\text{CoT}}(s')])}_{\Delta(s)}. \end{aligned}$$

By Assumption 1, the conditional distributions inside  $\mathcal{S}_{ok}$  are close in total variation, hence (by standard TV–expectation inequalities) for some constant  $C' > 0$ ,

$$|\mathbb{E}_{s' \sim T_m}[V_m(s')] - \mathbb{E}_{s' \sim T_{CoT}}[V_{CoT}(s')]| \leq C' \beta_{ok}.$$

Since  $0 \leq 1 - p_{\text{done}}(s) - \varepsilon_m(s) \leq 1$ , we have  $|\Delta(s)| \leq C' \beta_{ok}$ . Taking expectations over  $s \sim d_{CoT,t}$  and summing from  $t = i$  to  $i + \ell - 1$ , the additive “shift” terms accumulate only along trajectories that have not yet absorbed, which contributes the factor  $w_t$ ; let  $C \geq C'$  absorb constants and the bound on  $w_t \leq 1$ . This yields the stated lower bound. The approximation claim follows by dropping  $\Delta(s)$ , which is precisely the small conditional-shift term bounded via  $\beta_{ok}$ .

This leaves an approximation where the advantage is primarily driven by the reduction in the probability of making a catastrophic error. This proposition is directly supported by the empirical results in Figure-3, which shows a strong correlation between a module’s rank and its per-step error rate.  $\square$

### A.3 THEORETICAL GROUNDING FOR THE DECOUPLED META-POLICY SEARCH

We now justify the zero-shot transfer of a meta-policy  $\pi^*$  discovered using the surrogate  $m_{CoT}$  to the final module  $m^*$ . The goal is to show that the expected value of the full system improves, i.e.,

$$V(\pi^*(m^*)) \geq V(\pi^*(m_{CoT})).$$

**Proposition 2** (Module improvement on baseline states). *The scaffolded meta policy objective optimizes for a module  $m^*$  that has a non-negative expected advantage over the states induced by the baseline policy:*

$$\mathbb{E}_{s \sim d_{m_{CoT}}}[V_{m^*}(s)] \geq \mathbb{E}_{s \sim d_{m_{CoT}}}[V(s)].$$

**Proposition 3** (Beneficial distribution shift). *A superior module  $m^*$ , which has a lower error rate on-expectation compared to the baseline (i.e.,  $\mathbb{E}_{s \sim d_{m_{CoT}}}[\epsilon_{m^*}(s)] \leq \mathbb{E}_{s \sim d_{m_{CoT}}}[\epsilon_{m_{CoT}}(s)]$ ), induces a stationary state distribution  $d_{m^*}$  that is weighted towards higher-value states.*

$$\mathbb{E}_{s \sim d_{m^*}}[V_{m^*}(s)] \geq \mathbb{E}_{s \sim d_{CoT}}[V_{m^*}(s)]$$

- **Remark:** Reducing per-step hazard ( $S_{fail}$  states) increases expected survival time in  $\mathcal{S}_{ok}$ , shifting probability away from  $S_{fail}$ . Since  $V_{m^*}(s)$  is larger on  $\mathcal{S}_{ok}$  than on  $S_{fail}$ , the expected value under  $d_{m^*}$  is weakly greater.

We now justify the zero-shot transfer of a meta-policy  $\pi^*$  discovered using the surrogate  $m_{CoT}$  to the final module  $m^*$ . We aim to show that the expected value of the system improves:  $\overline{V}_{m^*} \geq \overline{V}_{m_{CoT}}$ . This transfer relies on two premises.

**Theorem 1** (Monotonic Improvement of Meta-Policy Transfer). *Let  $\pi(m)$  denote the system using meta-policy  $\pi$  and step-generator  $m$ , and let  $V(\pi(m))$  be its expected reward from the initial state distribution. If Proposition 2 and Proposition 3 hold, then the transfer of a meta-policy  $\pi^*$  from  $m_{CoT}$  to  $m^*$  is guaranteed to not degrade performance:*

$$V(\pi^*(m^*)) \geq V(\pi^*(m_{CoT})).$$

**Proof Sketch:** We can decompose the difference in expected values as follows:

$$V(\pi^*(m^*)) - V(\pi^*(m_{CoT})) = \mathbb{E}_{s \sim d_{m^*}}[V_{m^*}(s)] - \mathbb{E}_{s \sim d_{m_{CoT}}}[V(s)].$$

We can add and subtract the term  $\mathbb{E}_{s \sim d_{m_{CoT}}}[V_{m^*}(s)]$  to obtain:

$$\begin{aligned} V(\pi^*(m^*)) - V(\pi^*(m_{CoT})) &= \underbrace{\left( \mathbb{E}_{s \sim d_{m^*}}[V_{m^*}(s)] - \mathbb{E}_{s \sim d_{m_{CoT}}}[V_{m^*}(s)] \right)}_{\text{Gain from Distribution Shift}} \\ &\quad + \underbrace{\left( \mathbb{E}_{s \sim d_{m_{CoT}}}[V_{m^*}(s)] - \mathbb{E}_{s \sim d_{m_{CoT}}}[V(s)] \right)}_{\text{Gain from Module Improvement}}. \end{aligned}$$

The first term (*Gain from Distribution Shift*) is non-negative by Proposition-3. The second term (*Gain from Module Improvement*) is non-negative by Proposition-2. Since both terms are non-negative, their sum is non-negative, proving the theorem.

This theoretical decomposition is empirically validated by the experiment in Figure 3. The performance of *CoT→Meta* relative to *CoT Baseline* measures the Gain from Module Improvement, while the additional gain seen in the full *Meta Policy* system reflects the Gain from Distribution Shift. The positive results for both components provide strong empirical support for this theoretical justification.

## B STEP GENERATOR ROLL OUT EXAMPLES

### B.1 EXAMPLE-1

#### ARM Rollout

```

### 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 \(\sin 75^\circ = \sin(45^\circ + 30^\circ)\):
\[
\sin 75^\circ = \sin 45^\circ \cos 30^\circ + \cos 45^\circ \sin 30^\circ
\quad
\cos 75^\circ = \frac{\sqrt{2}}{2} \times \frac{\sqrt{3}}{2} + \frac{\sqrt{2}}{2} \times \frac{1}{2} = \frac{\sqrt{6}}{4} + \frac{\sqrt{2}}{4} = \frac{\sqrt{6} + \sqrt{2}}{4}.
\]
Hence,
\[
c = 2R \times \frac{\sqrt{6} + \sqrt{2}}{4} = R \times \frac{\sqrt{6} + \sqrt{2}}{2}.
\]
```

**CoT Rollout**

```

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

```

**ARM Rollout**

```

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

```

**CoT Rollout**

```

### Step 3
Determine the new temperature after doubling the kinetic energy:
Since kinetic energy \(\propto T\),
\[
T_2 = 2 \times T_1 = 526.3 \text{ K}
\]
Convert this back to Celsius:
\[
T_2 (\text{ }^\circ \text{C}) = T_2 (\text{K}) - 273.15 = 526.3 - 273.15 \approx 253.15^\circ \text{ C}
\]
This temperature is well above the boiling point of water at 1 atm
(100°C). 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).

```

### B.3 EXAMPLE-3

#### ARM Rollout

##### ### 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  $\langle P^+ \rangle$  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  $\langle P^+ \rangle$  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:

- $\langle P^+ \rangle$ : light-cone momentum component, usually  $\langle P^+ = P^0 + P^3 \rangle$ .
- $\langle \gamma^u \rangle$ : a gamma matrix with a specific index  $\langle u \rangle$ , possibly a spatial index.
- $\langle \gamma^0 \rangle$ : the gamma matrix corresponding to time.
- $\langle \gamma^0 P \rangle$ : a product of  $\langle \gamma^0 \rangle$  with some momentum operator  $\langle P \rangle$ .

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.

## CoT

```
### Step 1
```

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

Given the options:

- A:  $\langle P^+ \rangle$
- B:  $\langle \gamma^u \rangle$
- C:  $\langle \gamma^0 \rangle$
- D:  $\langle \gamma^0 P \rangle$

and based on the typical transformations, the substitution from  $\langle P^\mu \rangle$  to a form involving gamma matrices usually involves  $\langle \gamma^0 \rangle$  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  $\langle P^\mu \rangle$ , 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  $\langle P^0 \rangle$ .

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.

## C BEST ARM DISCOVERED: CRITICCHAINOFTHOUGHTV7

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

```

1 import asyncio
2
3 class CriticChainOfThoughtV7:
4     def __init__(self, llm):
5         self.llm = llm
6
7     async def forward(self, problem, partial_progress):
8         # 1. Generate four candidate next steps in parallel
9         candidate_tasks = [
10             self.llm.generate_step(problem, partial_progress)
11             for _ in range(4)
12         ]
13         candidates = await asyncio.gather(*candidate_tasks)
14
15         # 2. Critique candidates in two groups of two, in parallel
16         critique_tasks = []
17         groups = [
18             (0, 2, ("rating_1", "rating_2"), ("critique_1",
19             "critique_2")),
20             (2, 4, ("rating_3", "rating_4"), ("critique_3",
21             "critique_4"))
22         ]
23         for start, end, rating_names, critique_names in groups:
24             context = [
25                 {
26                     "name": "Problem",
27                     "data": problem,
28                     "description": "The problem to solve."
29                 },
30                 {
31                     "name": "Partial Progress",
32                     "data": partial_progress,
33                     "description": "The partial solution so far."
34                 },
35                 {
36                     "name": "Candidate Next Steps",
37                     "data": "\n\n".join(
38                         f"### Candidate Next Step\n"
39                         f"\t{i+1}\n{candidates[i]}\n"
40                         for i in range(start, end)
41                     ),
42                     "description": "Two candidate next steps\n"
43                     "formatted with markdown subheaders."
44                 }
45             ]
46             instructions = (
47                 "You are given a problem, the current partial\n"
48                 "solution, and two candidate next reasoning steps.\n"
49                 "For each candidate, provide:\n"
50                 f"- {rating_names[0]} and {rating_names[1]}: a single\n"
51                 "integer rating from 1 to 10 indicating its fit as the next\n"
52                 "reasoning step (10 is best).\n"
53                 f"- {critique_names[0]} and {critique_names[1]}: a\n"
54                 "one-sentence critique highlighting each candidate's strengths\n"
55                 "and weaknesses.\n"
56                 f"Name the fields exactly {rating_names[0]},\n"
57                 f"\t{rating_names[0]}, {rating_names[1]}, {critique_names[1]}."
58             )
59             response_format = [
60                 {

```

```

51             "name": rating_names[0],
52             "description": f"Integer rating (1-10) for
53     ↪ Candidate Next Step {start+1}."}
54         },
55         {
56             "name": critique_names[0],
57             "description": f"One-sentence critique of
58     ↪ Candidate Next Step {start+1}."}
59         },
60         {
61             "name": rating_names[1],
62             "description": f"Integer rating (1-10) for
63     ↪ Candidate Next Step {start+2}."}
64         },
65         {
66             "name": critique_names[1],
67             "description": f"One-sentence critique of
68     ↪ Candidate Next Step {start+2}."}
69         }
70     ]
71     critique_tasks.append(
72         self.llm.chat_completion(context, instructions,
73     ↪ response_format)
74     )
75
76 critiques = await asyncio.gather(*critique_tasks)
77
78 # 3. Parse ratings and identify the two highest-rated
79 ↪ candidates
80     ratings = [
81         int(critiques[0]["rating_1"]),
82         int(critiques[0]["rating_2"]),
83         int(critiques[1]["rating_3"]),
84         int(critiques[1]["rating_4"])
85     ]
86     sorted_indices = sorted(range(4), key=lambda i: ratings[i],
87     ↪ reverse=True)
88     top1_idx, top2_idx = sorted_indices[0], sorted_indices[1]
89     top1_candidate = candidates[top1_idx]
90     top2_candidate = candidates[top2_idx]
91
92 # 4. Final head-to-head comparison between the top two
93 ↪ candidates
94     context_final = [
95         {
96             "name": "Problem",
97             "data": problem,
98             "description": "The problem to solve."
99         },
100        {
101            "name": "Partial Progress",
102            "data": partial_progress,
103            "description": "The partial solution so far."
104        },
105        {
106            "name": "Candidate Next Steps",
107            "data": (
108                f"### Candidate A\n{n{top1_candidate}}\n\n"
109                f"### Candidate B\n{n{top2_candidate}}"
110            ),
111            "description": "Two top candidate next steps
112     ↪ formatted with markdown subheaders."
113        }
114    ]
115    instructions_final = (

```

```

107     "Compare Candidate A and Candidate B as the next
108     → reasoning step for the given problem and partial progress.\n"
109         "Provide:\n"
110             "- winner: choose either 'Candidate A' or 'Candidate B'"
111             "- justification: one-sentence justification for your
112             choice."
113         )
114         response_format_final = [
115             {
116                 "name": "winner",
117                 "description": "Either 'Candidate A' or 'Candidate B'"
118             },
119             {
120                 "name": "justification",
121                 "description": "One-sentence justification for the
122             choice."
123             }
124         ]
125         final_decision = await self.llm.chat_completion(
126             context_final, instructions_final, response_format_final
127         )
128
129     if final_decision["winner"].strip() == "Candidate A":
130         selected_candidate = top1_candidate
131         runnerup_candidate = top2_candidate
132     else:
133         selected_candidate = top2_candidate
134         runnerup_candidate = top1_candidate
135
136     # 5. Post-selection adversarial critique with severity rating
137     context_flaw = [
138         {
139             "name": "Problem",
140             "data": problem,
141             "description": "The problem to solve."
142         },
143         {
144             "name": "Partial Progress",
145             "data": partial_progress,
146             "description": "The partial solution so far."
147         },
148         {
149             "name": "Selected Candidate Next Step",
150             "data": f"## Selected Candidate Next
151             Step\n{selected_candidate}",
152             "description": "The final chosen candidate next
153             reasoning step formatted with a markdown subheader."
154         }
155     ]
156     instructions_flaw = (
157         "You are given a problem, the current partial solution,
158         and a selected next reasoning step.\n"
159         "Identify any major flaw or missing piece of reasoning in
160         the selected step.\n"
161         "Provide:\n"
162             "- flaw: either the single word 'None' if there is no
163             flaw, or a brief description of the flaw.\n"
164             "- severity: a single integer rating from 1 to 10
165             indicating how severe the flaw is (10 is critical)."
166         )
167         response_format_flaw = [
168             {
169                 "name": "flaw",
170                 "description": "The identified flaw or None if none exists."
171             }
172         ]

```

```

161         "description": "Either the single word 'None' if
162         ↪ there is no flaw, or a brief description of a major flaw in
163         ↪ the selected step."
164     },
165     {
166         "name": "severity",
167         "description": "Integer rating (1-10) indicating
168         ↪ severity of the flaw (10 is most severe)."
169     }
170 ]
171 flaw_response = await self.llm.chat_completion(
172     context_flaw, instructions_flaw, response_format_flaw
173 )
174 flaw = flaw_response["flaw"].strip()
175 severity = int(flaw_response["severity"])
176
177 # 6. Compute dynamic severity threshold based on rating gap
178 gap = ratings[top1_idx] - ratings[top2_idx]
179 if gap <= 1:
180     threshold = 5
181 elif gap == 2:
182     threshold = 6
183 else:
184     threshold = 7
185
186 # 7. If a severe flaw is detected above the dynamic
187 ↪ threshold, fall back
188 if flaw.lower() != "none" and severity >= threshold:
189     return runnerup_candidate
190 return selected_candidate

```

Listing 1: Code for CriticChainOfThoughtV7, performance: 38.0

## D BEST META-POLICY DISCOVERED: VERIFIEDWEIGHTEDADAPTIVESELFCONSISTENTCHAINOFTHOUGHT

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

```

1 import asyncio
2 from agent.solution import Solution, Step
3 from judge_utils import judge_equality
4
5 class VerifiedWeightedAdaptiveSelfConsistentChainOfThought:
6     def __init__(self, llm, block):
7         self.llm = llm
8         self.block = block
9
10    @async def forward(self, problem):
11        # Helper: generate one chain up to 8 steps, then complete via
12        ↪ LLM if needed
13        @async def generate_chain():
14            solution = Solution()
15            for _ in range(8):
16                next_step = await self.block.forward(problem,
17                ↪ str(solution))
18                solution.add_step(Step(str(next_step)))
19                if solution.is_completed():
20                    return solution
21            completion = await self.llm.complete_solution(problem,
22            ↪ str(solution))
23            solution.add_step(Step(str(completion)))
24        return solution

```

```

22
23     # Helper: confidence scoring (1-5)
24     async def score_chain(chain):
25         context = [
26             {"name": "Problem", "data": problem, "description": "
27             "The original problem statement."},
28             {"name": "Chain", "data": str(chain),
29             "description": "Full chain-of-thought reasoning plus final
30             answer."}
31         ]
32         instructions = (
33             "You are evaluating the chain-of-thought solution for
34             the given problem. "
35             "On a scale from 1 (very uncertain) to 5 (very
36             confident), rate your confidence "
37             "that the final answer is correct. Output ONLY the
38             integer confidence (1-5)."
39         )
40         response_format = [{"name": "Confidence", "description": "
41             "Integer from 1 to 5"}]
42         resp = await self.llm.chat_completion(context,
43             instructions, response_format)
44             # parse safely
45             try:
46                 conf = int(resp["Confidence"].strip())
47             except Exception:
48                 conf = 1
49             return max(1, min(conf, 5))

50
51     # Helper: verify logical consistency (Yes/No)
52     async def verify_chain(chain):
53         context = [
54             {"name": "Problem", "data": problem, "description": "
55             "The original problem statement."},
56             {"name": "Chain", "data": str(chain),
57             "description": "Full chain-of-thought reasoning plus final
58             answer."}
59         ]
60         instructions = (
61             "Review the chain-of-thought reasoning for the given
62             problem. "
63             "Is the reasoning free of logical errors or
64             contradictions? "
65             "Output ONLY 'Yes' if it is fully logical, otherwise
66             output 'No'."
67         )
68         response_format = [{"name": "Valid", "description": "Yes
69             or No"}]
70         resp = await self.llm.chat_completion(context,
71             instructions, response_format)
72         valid = resp.get("Valid",
73             "").strip().lower().startswith("y")
74         return valid

75
76     # Weighted vote helper
77     def find_best_weighted(chains_list, conf_list):
78         weight_sums = {}
79         total = sum(conf_list)
80         for chain, cf in zip(chains_list, conf_list):
81             ans = chain.answer()
82             weight_sums[ans] = weight_sums.get(ans, 0) + cf
83         best_ans, best_w = None, -1
84         for ans, w in weight_sums.items():
85             if w > best_w:
86                 best_ans, best_w = ans, w

```

```

70     return best_ans, best_w, total
71
72     # 1) Generate initial 3 chains in parallel
73     initial = [generate_chain() for _ in range(3)]
74     chains = await asyncio.gather(*initial)
75
76     # 2) Score and verify each chain
77     score_tasks = [score_chain(ch) for ch in chains]
78     verify_tasks = [verify_chain(ch) for ch in chains]
79     confidences = await asyncio.gather(*score_tasks)
80     valids = await asyncio.gather(*verify_tasks)
81
82     max_chains = 7
83
84     # 3) Adaptive sampling with verification gating
85     while True:
86         # Determine which chains to consider: only verified if
87         # any, else all
88         if any(valids):
89             considered_chains = [ch for ch, v in zip(chains,
90             valids) if v]
91             considered_confs = [cf for cf, v in
92             zip(confidences, valids) if v]
93         else:
94             considered_chains = chains
95             considered_confs = confidences
96
97             best_ans, best_weight, total_weight =
98             find_best_weighted(considered_chains, considered_confs)
99             # stop if weighted majority reached or chain cap
100            if best_weight > total_weight / 2 or len(chains) >=
101            max_chains:
102                break
103
104            # else generate one more chain, score & verify, then loop
105            new_chain = await generate_chain()
106            chains.append(new_chain)
107            new_conf = await score_chain(new_chain)
108            confidences.append(new_conf)
109            new_valid = await verify_chain(new_chain)
110            valids.append(new_valid)
111
112            # 4) Select final chain: consensus & highest confidence among
113            # considered
114            if any(valids):
115                final_pool = [ (ch, cf) for ch, cf, v in zip(chains,
116                confidences, valids) if v and judge_equality(ch.answer(),
117                best_ans) ]
118            else:
119                final_pool = [ (ch, cf) for ch, cf in zip(chains,
120                confidences) if judge_equality(ch.answer(), best_ans) ]
121
122            selected_chain = None
123            top_conf = -1
124            for ch, cf in final_pool:
125                if cf > top_conf:
126                    selected_chain, top_conf = ch, cf
127
128            # Fallback if nothing selected
129            if selected_chain is None:
130                selected_chain = chains[-1]
131
132            return selected_chain

```

Listing 2: Code for VerifiedWeightedAdaptiveSelfConsistentChainOfThought, performance: 38.0

## E 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 OpenAI’s 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.

### E.1 ARM IMPLEMENTATION DETAILS

The 1000-sample subset of Open-R1-Mixture-of-Thoughts was created by taking the math and science splits of the original dataset, filtering to samples which the provided Deepseek-R1 reasoning trace had length between 8k to 10k tokens (to filter to samples of appropriate difficulty), and randomly sampling 1000 problems from the filtered problems.

We run both the step-generator 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.

### E.2 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.