LEARNING ADAPTIVE REASONING THREADS FOR MATHEMATICAL PROBLEM SOLVING

Anonymous authors
Paper under double-blind review

ABSTRACT

Long chain-of-thought (CoT) has substantially improved mathematical reasoning by giving models more computation to develop and refine their solutions. Yet complex problem solving is not naturally linear: some intermediate steps may require extensive reasoning, while later steps need only their conclusions. Linear CoT keeps all such computation in a single growing context, where obsolete details, checks, and failed attempts can interfere with subsequent reasoning. We therefore ask whether a single LLM can learn to selectively develop intermediate derivations separately and reincorporate their conclusions when needed. We introduce Adaptive Reasoning Threads (ART), an *on-demand, state-driven* protocol that lets the model selectively extend reasoning from its evolving main-thread state into private local threads. Each extension inherits the problem and reasoning accumulated so far, continues with the same model policy, and returns only its conclusion when needed. We learn this behavior through supervised fine-tuning followed by outcome-based reinforcement learning. Across model scales and competition-mathematics benchmarks, ART consistently outperforms matched linear-CoT baselines, including longer-CoT controls with comparable total generation budgets. Simply exposing the ART interface is insufficient to reproduce these gains, showing that effective extension behavior must itself be learned. Moreover, ART-trained models remain stronger under standard CoT decoding and exhibit substantially fewer corrections and restarts. These results suggest that learning when to extend local reasoning can improve mathematical problem solving beyond simply increasing reasoning length.

1 INTRODUCTION

Recent advances in LLM reasoning have substantially improved performance on challenging mathematical problems. Modern reasoning models have advanced these capabilities by learning to use longer chain-of-thought (CoT) trajectories. These extended trajectories provide room for additional reasoning computation, allowing models to develop and refine their solutions more thoroughly before committing to a final answer (Guo et al., 2025; Team et al., 2025; Muennighoff et al., 2025).

However, solving complex problems is not inherently a linear process. Consider a mathematical proof: one may first form a high-level plan, then spend substantial effort proving an intermediate lemma in detail, while the remainder of the solution only needs the lemma's conclusion rather than the full derivation that produced it. Linear CoT is potentially suboptimal for such reasoning patterns because it forces every intermediate step into a single, ever-growing sequence. Detailed derivations, discarded attempts, verification steps, and even mistakes all remain in the context long after they have served their purpose. As they accumulate, this increasingly irrelevant reasoning can clutter the context, dilute attention to the information that matters for subsequent steps, and ultimately interfere with the model's ability to carry out the overall solution.

Recent work has begun to relax the linear constraint of CoT through several forms of structured reasoning. Parallel reasoning expands computation across multiple paths to promote broader exploration and verification (Pan et al., 2025; Zheng et al., 2026). Tree-based reasoning explicitly grows and searches over alternative reasoning trajectories (Yao et al., 2023; Guan et al., 2025). Trajectory-compression methods instead compress intermediate reasoning and interaction histories into compact outcomes to control active-context growth (Sun et al., 2025; Kang et al., 2025).Despite

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

their different forms, these approaches largely take a *structure-predefined* view, organizing reasoning around a prescribed structural objective such as broader exploration or a smaller active context. For challenging mathematical reasoning, we argue that the reasoning structure itself should instead *adapt* as the solution unfolds, emerging *on demand* from the model's evolving reasoning state.

From this perspective, we introduce Adaptive Reasoning Threads (ART), an *on-demand, state-driven* reasoning protocol that allows a single LLM to unfold its own evolving trajectory adaptively according to the reasoning state. ART is built around a single main thread, which remains the default path for developing the overall solution. When needed, the model can extend a local derivation into a private thread. Each private thread extends from the current point of the main trajectory and inherits both the original problem and the reasoning accumulated so far. It therefore continues from the same assumptions, notation, and intermediate results. The selected issue is developed in its own context, and only its conclusion is retrieved to the main thread when needed. We acquire this behavior through post-training: supervised fine-tuning establishes reliable ART reasoning patterns, while outcome-based reinforcement learning further adapts when to extend local computation and when to retrieve and use its result.

The design of ART keeps the main thread coherent while preventing detailed local computation from overwhelming its trajectory. As each private thread extends from the current main-thread state, it remains anchored to the solution developed so far. Ideally, this state inheritance becomes especially useful as reasoning deepens and more assumptions, notation, and valuable intermediate results accumulate along the main thread. Most reasoning therefore remains inline, with private threads serving only as temporary extensions when separate treatment is considered as useful.

To sum up, our contributions are summarized as follows:

- We identify a limitation of linear CoT for difficult mathematical reasoning. Complex solutions often involve a clear main line supported by deeper local derivations, while linear CoT accumulates both in the same growing context. This motivates allowing local reasoning to unfold *adaptively* from the evolving solution state.
- We introduce Adaptive Reasoning Threads (ART), an *on-demand, state-driven* framework that enables a single LLM to selectively unfold local reasoning from its evolving trajectory. ART keeps a main thread for the overall solution and selectively extends local computation into private threads from the current reasoning state. These threads inherit the reasoning accumulated so far and return only their conclusions, preserving a clear main thread while isolating detailed local computation.
- We show that learned ART consistently improves mathematical reasoning across model scales and competition benchmarks. The gains persist under matched reasoning budgets and cannot be obtained by merely exposing the extension interface. Moreover, ART-trained models remain stronger under standard CoT decoding and exhibit substantially fewer corrections and restarts, suggesting that learning when to extend local reasoning also improves how the main reasoning trajectory itself unfolds.

2 Related Work

Mathematical reasoning with LLMs. Chain-of-thought (CoT) prompting improves mathematical problem solving by eliciting intermediate reasoning steps (Wei et al., 2022), while trained verifiers improve answer selection from sampled solutions (Cobbe et al., 2021). More recent work strengthens mathematical reasoning through domain-specific training and reinforcement learning (Shao et al., 2024; Guo et al., 2025). Long-context scaling and inference-budget control further enable models to use extended computation to develop and refine their solutions (Team et al., 2025; Muennighoff et al., 2025). However, research on overthinking shows that longer reasoning can introduce redundant computation with limited gains in accuracy (Chen et al., 2024). ART builds on these advances by studying whether post-training can teach a model when to extend local reasoning from its evolving solution state. The goal is to preserve the benefits of additional computation without allowing detailed local derivations to overwhelm the main trajectory.

Structured reasoning. Structured reasoning methods organize computation through decomposition, multiple paths, and explicit search. Decomposition methods break problems into simpler subproblems (Zhou et al., 2022), while self-consistency aggregates answers from independently

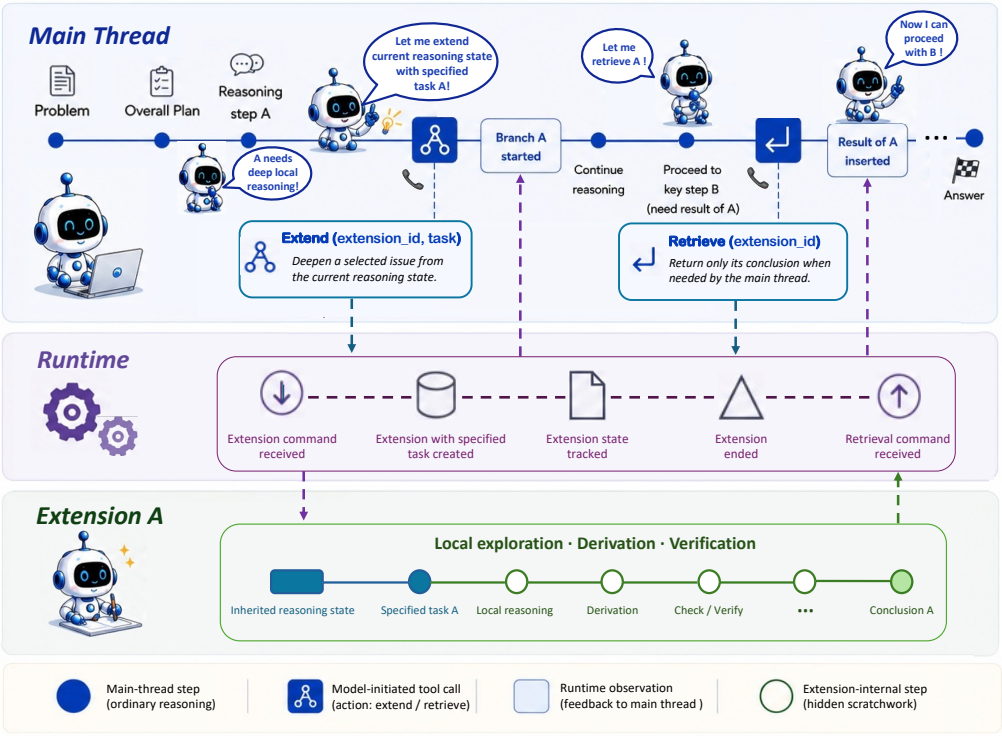

Figure 1: **Overview of Adaptive Reasoning Threads (ART).** ART allows a single LLM to unfold local reasoning on demand as its main trajectory evolves. Selected local computation is developed in private extensions from the current reasoning state, while only their conclusions are later reincorporated into the main trajectory. The resulting extension structure is determined online from the evolving reasoning state rather than fixed by a predetermined decomposition.

sampled reasoning paths (Wang et al., 2022). Tree- and graph-based methods explore, evaluate, or combine intermediate reasoning states (Yao et al., 2023; Besta et al., 2024; Guan et al., 2025). More recent work learns how reasoning paths are organized: APR coordinates serial and parallel threads to improve inference efficiency (Pan et al., 2025), while Parallel-R1 learns multi-path exploration and verification through post-training (Zheng et al., 2026). Context-Folding instead learns to manage context growth by folding completed sub-trajectories into concise outcomes (Sun et al., 2025). These approaches largely take a *structure-first* view, organizing reasoning around broader exploration, parallel execution, or a smaller active context. ART instead allows extension structure to emerge *on demand* from the evolving reasoning state. The main thread remains the default, while the model learns when to unfold local computation from the current reasoning state into a private extension and when to reincorporate its conclusion.

3 Adaptive Reasoning Threads

3.1 Overview

ART allows a single LLM to organize its reasoning into a main thread and private local extensions, supported by a lightweight runtime, as illustrated in Fig. 1. The main thread remains the default path for developing the overall solution. When the model judges that a local issue needs deeper reasoning, it issues an `extend` tool call to the runtime. The runtime then instantiates a private extension from the current main-thread state. This extension inherits the original problem and the reasoning accumulated so far. The same model policy then develops the selected issue in a separate context. The main thread can continue developing the solution and issue a `retrieve` tool call when the local conclusion is needed. The runtime then returns only the conclusion, helping prevent lengthy local details from distracting the model or contaminating the main context.

3.2 Protocol of Adaptive Reasoning Threads

3.2.1 Problem formulation

Given a mathematical problem x, ART uses a shared policy π_θ for both main-thread and extension reasoning. Only the main thread produces the final answer to x. Let t index main-thread generation segments and M_t denote the context before segment t, including preceding reasoning and tool interactions. The model generates the next segment u_t, consisting of ordinary reasoning or a control call, as

$$u_t \sim \pi_\theta(\cdot \mid x, M_t). \tag{1}$$

The generated segment and any subsequent runtime response o_t are appended to the main-thread context:

$$M_{t+1} = M_t \oplus u_t \oplus o_t, \tag{2}$$

where $\oplus$ denotes concatenation and o_t is empty when no runtime response is returned.

3.2.2 Control actions

In ART, we provides two trajectory control actions, `extend` and `retrieve`, alongside ordinary main-thread reasoning. Both are implemented through the model's standard tool-calling interface, enabling the model to unfold its own reasoning on demand.

Extend action. At step t_i, the model can issue an $\texttt{extend}(i, \tau_i)$ call. Here, i identifies the extension and τ_i specifies the local issue to develop. The runtime creates a private extension conditioned on the original problem and the main-thread context at the point of the call. Using the same policy, the extension generates a local reasoning trace R_i and a conclusion z_i:

$$(R_i, z_i) \sim \pi_\theta(\cdot \mid x, M_{t_i}, \tau_i). \tag{3}$$

The inherited state M_{t_i} provides the solution context, while τ_i specifies what to develop further. The `extend` operation thus enables a private local continuation of the model's own reasoning from the current point of the main thread. This allows deeper exploration of the selected issue without adding its intermediate drafts to the main trajectory.

Retrieve action. When a local conclusion is needed to advance the main solution, the model issues a $\texttt{retrieve}(i)$ call in a later turn. Only the conclusion z_i is incorporated into the current main-thread context. The detailed trace R_i remains private. This allows the model to build on its own local continuation without carrying its intermediate drafts back into the main thread.

3.3 Learning Adaptive Reasoning Threads

The ART protocol defines how local reasoning can unfold from the main trajectory, while post-training teaches the model to use this capability adaptively. We learn ART in two stages: ART-SFT establishes a reliable state-conditioned extension–retrieval pattern, and ART-GRPO further learns when local computation should be extended and reincorporated through self-generated rollouts and outcome feedback.

3.3.1 ART-SFT

ART-SFT serves as a cold-start stage that teaches the model to reliably execute the ART reasoning pattern before learning when local extension is useful. Each training example contains a complete ART trajectory, including main-thread reasoning, `extend`/`retrieve` actions, and private extension reasoning conditioned on the corresponding main-thread state and local objective. We apply the SFT loss to all model-generated spans in both the main thread and extensions, while treating prompts and runtime observations only as context. The same policy is shared across both reasoning contexts.

3.3.2 ART-GRPO

We optimize ART with GRPO on self-generated rollouts, using outcome feedback to adapt how the main trajectory invokes and uses local computation.

Main-thread GRPO objective. In ART, adaptive reasoning organization is realized through decisions made on the main thread: the model decides when to extend local computation, what to develop, and when to reincorporate its conclusion into the evolving solution. We therefore apply direct GRPO credit only to model-generated main-thread tokens, including ordinary reasoning and `extend`/`retrieve` calls. Prompts, runtime observations, retrieved conclusions, and extension-private reasoning are masked from the policy loss.

Following standard GRPO, let A_k denote the group-normalized advantage of rollout k. For token $u_{k,\ell}$ generated under context $c_{k,\ell}$, we define the policy ratio as

$$\rho_{k,\ell}(\theta) = \frac{\pi_\theta(u_{k,\ell} \mid c_{k,\ell})}{\pi_{\theta_{\text{old}}}(u_{k,\ell} \mid c_{k,\ell})}. \tag{4}$$

To implement this credit assignment, we introduce a mask $m^{\text{main}}_{k,\ell}$, which is 1 for model-generated main-thread tokens and 0 otherwise. For a group of K ART rollouts, we optimize

$$\mathcal{L}_{\text{rl}}(\theta) = -\mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K}\frac{\sum_{\ell=1}^{L_k} m^{\text{main}}_{k,\ell}\min(\rho_{k,\ell}A_k,\ \text{clip}(\rho_{k,\ell}, 1-\epsilon, 1+\epsilon)A_k)}{\sum_{\ell=1}^{L_k} m^{\text{main}}_{k,\ell}}\right]. \tag{5}$$

We additionally apply a standard KL penalty against a frozen reference policy. Masking extension-private tokens from the policy loss does not isolate extension reasoning from learning. The same policy is shared across main-thread and extension contexts, while extension conclusions affect subsequent main-thread reasoning and the resulting rollout reward.

Outcome reward with ART completion shaping. The primary learning signal is correctness on the original problem. However, under outcome-only RL, the newly acquired ART behavior can quickly collapse back to linear CoT before the policy learns how to organize local computation effectively. We therefore introduce a lightweight shaping reward to preserve complete extension–retrieval behavior during optimization.

For rollout k, we define

$$r_k = r_k^{\text{task}} + r_k^{\text{art}}, \tag{6}$$

where

$$r_k^{\text{task}} = \begin{cases} +1, & \text{if the final boxed answer is correct,} \\ -1, & \text{otherwise,} \end{cases} \tag{7}$$

and

$$r_k^{\text{art}} = \begin{cases} +0.1, & \text{completed extension–retrieval interaction,} \\ -0.05, & \text{attempted but incomplete interaction,} \\ -0.2, & \text{genuine no-extension rollout,} \\ 0, & \text{indeterminate due to runtime failure.} \end{cases} \tag{8}$$

A rollout is considered complete if it successfully creates at least one extension, retrieves all created extensions, and terminates without protocol errors or truncated extension traces. Importantly, the shaping reward does not increase with the number of extensions: one complete interaction receives the same bonus as several. It therefore encourages the policy to retain ART as a usable reasoning behavior without directly favoring larger extension structures. The no-extension penalty similarly discourages complete collapse back to linear CoT, while final-answer correctness remains the primary signal for learning which organization strategies are useful.

3.4 Discussion

ART's adaptive unfolding relies on two complementary design principles. The resulting extension structure is determined online from the evolving reasoning state rather than fixed by a predetermined decomposition, while the main thread remains the persistent state of the overall solution. We discuss how state-conditioned extensions and a persistent main thread support these two properties.

State-conditioned extension. The need for deeper local reasoning typically arises only after the model has established relevant assumptions, notation, and intermediate results. Conditioning an extension on the current main-thread state allows the same policy to continue reasoning directly

Table 1: Main results on competition-mathematics benchmarks with $N=16$ samples. Each training recipe is evaluated under both CoT and ART inference. Shaded rows denote ART-trained models, with the darker shade marking the final ART configuration. Within each phase, the best result is in **bold** and the second-best is underlined.

			AIME'25		AIME'26		HMMT'26		Average	
Model	Training	Inference	mean@16	pass@16	mean@16	pass@16	mean@16	pass@16	avg@16	pass@16
Qwen3.5-4B	Baseline	CoT	22.1	50.0	17.5	56.7	13.8	39.4	17.8	48.7
		ART	12.5	63.3	12.1	53.3	11.9	39.4	12.2	52.0
	Phase I: Cold-start SFT									
	CoT-SFT	CoT	51.5	80.0	51.0	73.3	29.9	60.6	44.1	71.3
		ART	48.5	70.0	47.9	76.7	26.7	60.6	41.0	69.1
	ART-SFT	CoT	52.3	76.7	54.2	76.7	34.8	60.6	47.1	71.3
		ART	**52.7**	**83.3**	**54.4**	**86.7**	**36.6**	**66.7**	**47.9**	**78.9**
	Phase II: RL									
	CoT-SFT+GRPO	CoT	66.2	83.3	74.4	93.3	50.6	78.8	63.7	85.1
		ART	66.9	86.7	70.2	86.7	44.1	75.8	60.4	83.1
	ART-SFT+ART-GRPO	CoT	68.3	**90.0**	73.3	90.0	50.2	75.8	63.9	85.3
		ART	**71.2**	**90.0**	**75.8**	**93.3**	**53.4**	**81.8**	**66.8**	**88.4**
Qwen3.5-9B	Baseline	CoT	25.8	53.3	18.3	46.7	15.0	30.3	19.7	43.4
		ART	23.1	63.3	26.5	66.7	24.4	51.5	24.7	60.5
	Phase I: Cold-start SFT									
	CoT-SFT	CoT	58.3	80.0	56.9	76.7	38.3	63.6	51.2	73.4
		ART	53.8	80.0	53.8	76.7	27.7	**72.7**	45.1	76.5
	ART-SFT	CoT	62.5	83.3	62.3	**83.3**	43.0	69.7	55.9	78.8
		ART	**63.3**	**93.3**	**67.1**	**83.3**	**43.8**	69.7	**58.1**	**82.1**
	Phase II: RL									
	CoT-SFT+GRPO	CoT	73.8	83.3	77.5	93.3	54.2	78.8	68.5	85.1
		ART	66.9	90.0	66.0	93.3	44.5	72.7	59.1	85.3
	ART-SFT+ART-GRPO	CoT	74.0	**93.3**	79.0	**96.7**	**58.7**	**81.8**	70.6	**90.6**
		ART	**74.8**	**93.3**	**81.5**	93.3	**58.7**	78.8	**71.7**	88.5

from the point where the local need emerges, rather than reconstructing the relevant context from the original problem. An extension is therefore not a standalone subproblem specified before reasoning begins, but a continuation instantiated from the state that gives rise to it. As a result, the placement, number, and content of extensions can be decided online according to the evolving solution.

Persistent main thread. While local computation can unfold adaptively, the main thread preserves the continuous state of the overall solution. Extensions serve as temporary computation from selected points along this trajectory, and only their conclusions are reincorporated when needed. This separation allows supporting computation to remain transient without fragmenting the persistent solution state across multiple evolving trajectories. Their detailed traces remain outside the persistent context. This asymmetry also motivates our main-thread-centered optimization, which assigns direct credit to the trajectory that decides when to extend and how to use the returned conclusions.

4 Experiments

4.1 Experimental Setup

Data and training. For supervised cold-start training, we construct a 961-problem set from AIME 1983–2024 and DeepScaleR (Luo et al., 2025). A strong teacher generates trajectories under the corresponding reasoning protocol, from which we retain correct solutions and, for ART, additionally remove trajectories with protocol errors. Reinforcement learning uses a separate 9,117-problem pool filtered from DAPO-Math (Yu et al., 2026) and DeepMath (He et al., 2025), with rollouts sampled on-policy rather than supplied by a teacher. We train Qwen3.5-4B and Qwen3.5-9B (Qwen Team, 2026), applying SFT for three epochs followed by 250 GRPO steps. The matched CoT and ART settings use the same problem pools, teacher model, and training schedule.

Evaluation. We evaluate on AIME 2025 (Zhang and Math-AI, 2025), AIME 2026 (Zhang and Math-AI, 2026), and HMMT February 2026 (Dekoninck et al., 2026), denoted HMMT'26 throughout the paper. For each problem, we sample $N=16$ solutions and report mean@16 and pass@16, together with ART-specific measures of extension use, protocol failure, and retrieval completion. All evaluations use thinking mode with a maximum response length of 16,384 tokens. Complete data construction, metric definitions and training and decoding settings are provided in Sec. A.

Table 2: Adding a Python code_interpreter tool during RL, with no Python-specific cold-start SFT. The Average column is the unweighted mean over AIME'25, AIME'26, and HMMT'26.

			AIME'25		AIME'26		HMMT'26		Average	
Model	Training	Inference	mean@16	pass@16	mean@16	pass@16	mean@16	pass@16	avg@16	pass@16
Qwen3.5-4B	CoT-SFT + GRPO	CoT	66.2	83.3	74.4	93.3	50.6	78.8	63.7	85.1
	+ code interpreter	CoT	68.1	86.7	73.5	90.0	47.3	78.8	63.0	85.2
	ART-SFT + ART-GRPO	ART	71.2	90.0	75.8	93.3	53.4	81.8	66.8	88.4
	+ code interpreter	**ART**	**75.4**	**100.0**	**84.0**	**96.7**	**65.3**	**90.9**	**74.9**	**95.9**
Qwen3.5-9B	CoT-SFT + GRPO	CoT	73.8	83.3	77.5	93.3	54.2	78.8	68.5	85.1
	+ code interpreter	CoT	72.9	86.7	77.9	96.7	50.8	84.8	67.2	89.4
	ART-SFT + ART-GRPO	ART	74.8	93.3	81.5	93.3	58.7	78.8	71.7	88.5
	+ code interpreter	**ART**	**84.4**	**100.0**	**91.9**	**100.0**	**70.5**	**90.9**	**82.3**	**97.0**

Table 3: ART behavior across training stages under ART-mode inference, averaged over AIME'25, AIME'26, and HMMT'26. *Extension use* denotes the fraction of rollouts that successfully create at least one local extension; *retrieval completion* denotes the fraction of created extensions whose conclusions are successfully retrieved. The final ART model is highlighted.

Model	Method	Extension use (%)	Protocol fail (%) ↓	Retrieval completion (%) ↑	Acc.↑
Qwen3.5-4B	Baseline	8.2	30.7	70.6	12.2
	Phase I: Cold-start SFT				
	CoT-SFT	4.6	5.8	95.7	41.0
	ART-SFT	29.2	0.2	99.8	47.9
	Phase II: RL				
	CoT-SFT + GRPO	6.3	19.8	83.0	60.4
	ART-SFT + ART-GRPO	77.3	2.1	98.7	66.8
Qwen3.5-9B	Baseline	21.5	12.4	89.1	24.7
	Phase I: Cold-start SFT				
	CoT-SFT	8.5	6.9	93.6	45.1
	ART-SFT	34.5	0.6	99.6	58.1
	Phase II: RL				
	CoT-SFT + GRPO	13.4	11.3	90.2	59.1
	ART-SFT + ART-GRPO	75.3	0.5	99.7	71.7

4.2 Main Results: Does Learned ART Improve Reasoning?

Learned ART improves reasoning. As shown in Tab. 1, the full ART pipeline consistently outperforms the matched CoT pipeline across both model scales and all three competition-mathematics benchmarks. After Phase II RL, average mean@16 increases from 63.7 to 66.8 for Qwen3.5-4B and from 68.5 to 71.7 for Qwen3.5-9B. These results show that learning to selectively develop demanding local issues outside the main trajectory can improve overall problem-solving performance.

The gain comes from learning the control, not exposing the interface. Cross-evaluating training recipes and inference modes separates the effect of the ART interface from the behavior learned through post-training. Without ART-specific training, simply exposing baseline or CoT-trained models to the extension interface is unreliable and often reduces accuracy. In contrast, ART-trained models remain strong under ordinary CoT decoding, and enabling ART inference on the same models provides a further gain: average mean@16 rises from 63.9 to 66.8 at 4B and from 70.6 to 71.7 at 9B. Thus, the benefit does not come from access to additional reasoning paths alone; it depends on learning how local extensions should be introduced and reincorporated within an evolving trajectory.

The learned control extends to external computation. We next ask whether ART is tied specifically to language-only local reasoning. During Phase II RL, we additionally provide a Python `code_interpreter`, without any Python-specific cold-start supervision. The tool is available to both the main trajectory and local extensions, while extensions remain leaf computations and cannot create further extensions. As shown in Tab. 2, adding the same tool to the matched CoT pipeline provides little benefit, whereas ART with Python raises average mean@16 from 66.8 to 74.9 at 4B and from 71.7 to 82.3 at 9B. This suggests that ART can use the same learned reasoning control to incorporate additional forms of local computation when they become available. The complete prompt augmentation, tool schema, and an example trajectory are provided in Sec. C.4.

4.3 Extension Behavior: How Does ART Learn Adaptive Extension?

Post-training stages. We examine how ART behavior evolves across post-training stages in Tab. 3. ART-SFT primarily establishes reliable extension–retrieval interactions, reducing protocol failures

Table 4: Reward-shaping ablation for ART-GRPO on Qwen3.5-4B, evaluated on HMMT'26. *ART completion* gives reward for a completed EXTEND–RETRIEVE interaction and penalty for an attempted but incomplete one. *Ext.-count bonus* additionally rewards the number of successfully created extensions, whereas *zero-ext. penalty* acts only when no extension is attempted.

Config	Reward shaping			HMMT'26			
	ART completion	Ext.-count bonus	Zero-ext. penalty	mean@16	pass@16	Extension use (%)	Ext./used rollout
Outcome only	0	0	0	49.2	75.8	0.2	1.00
+ ART completion	$+0.1/-0.05$	0	0	50.2	78.8	51.1	2.00
+ extension-count bonus	$+0.1/-0.05$	$+0.2$	0	50.7	72.7	65.2	3.84
+ zero-extension penalty **(ours)**	$+0.1/-0.05$	0	-0.2	**53.4**	**81.8**	56.8	2.06

to nearly zero and raising retrieval completion to about 99%. ART-GRPO then makes this capability more broadly used during problem solving: extension use increases from 29.2% to 77.3% at 4B and from 34.5% to 75.3% at 9B, while retrieval completion remains near 99% and accuracy reaches its highest level. Importantly, conditioned on extension use, the average number of extensions increases only modestly from 1.76/1.73 to 2.12/2.03, suggesting that RL primarily broadens when local extension is exercised rather than indiscriminately increasing extensions.

Reward shaping. Fig. 2 and Tab. 4 clarify the role of our reward shaping. Under outcome-only optimization, extension use rapidly collapses toward zero, whereas ART-completion shaping keeps complete EXTEND–RETRIEVE interactions available during RL. As a quantity-driven alternative, we add an extension-count bonus $0.2\min(N_{\text{ext}}, 3)$, where N_{ext} is the number of successfully created extensions. This increases extension use to 65.2% and conditional fan-out to 3.84, but provides little improvement in mean@16. Our zero-extension penalty instead acts only when the extension option disappears entirely, achieving the best accuracy while keeping the number of extensions per used rollout close to completion shaping alone (2.06 vs. 2.00). These results support allowing extension structure to emerge on demand from the evolving reasoning state, rather than prescribing it through a quantity-driven objective.

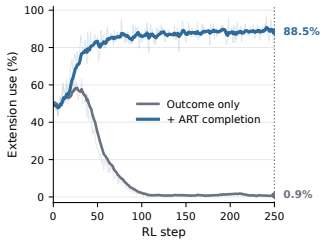

Figure 2: Extension use during ART-GRPO with and without ART-completion shaping.

Design ablations. Tab. 5 supports two key choices in ART. Without state inheritance, the model uses extensions more often but performs worse, suggesting that local reasoning becomes less reliable when it is detached from the evolving solution state and must compensate with additional extensions. All-token GRPO shows a similar trend: stronger extension use does not translate into better accuracy, indicating that assigning direct credit to both main-thread reasoning and temporary local computation can blur credit assignment. These results favor state-conditioned extension and main-thread-centered optimization over simply increasing local computation.

Table 5: Ablation of the core ART design choices on Qwen3.5-4B over HMMT'26.

Configuration	mean@16 ↑	Ext. use (%)	Ext./used
Full ART	**53.4**	56.8	2.06
w/o state inheritance	50.2	60.4	2.24
w/ all-token GRPO	51.5	63.1	2.31

4.4 Reasoning Pattern: How Does Learning ART Change Reasoning?

The learned organization transfers to ordinary CoT. ART provides an explicit setting for learning when demanding local reasoning should be separated from the main trajectory. We ask whether learning this distinction also changes how the model reasons when such separation is no longer available. We therefore compare CoT-SFT and ART-SFT under ordinary CoT decoding, with the extension interface removed. As shown in Tab. 6, ART-SFT produces substantially fewer correction markers, `Wait` markers, and explicit restarts across both model scales, while also achieving stronger CoT accuracy in Tab. 1. Since all reasoning must remain on a single trajectory at inference time, these differences suggest that learning to separate demanding local computation also changes how the main reasoning trajectory itself develops. Detailed metric definitions are provided in Sec. A.

Controlling for reasoning budget. As shown in Tab. 7, ART achieves higher accuracy than the matched CoT baseline while maintaining a shorter main trajectory at both model scales. To separate the effect of local extension from simply generating more tokens, we further compare against a longer-CoT control with a closely matched total generation budget; ART remains consistently

Table 6: Reasoning-style event frequencies under CoT inference, averaged over AIME'25, AIME'26, and HMMT'26.

Model	SFT data	Correction frequency ↓	`Wait` frequency ↓	Restart frequency ↓
Qwen3.5-4B	CoT	15.0	8.2	0.39
	ART	**9.4**	**5.7**	**0.03**
Qwen3.5-9B	CoT	14.5	7.8	0.38
	ART	**8.6**	**5.2**	**0.02**

Table 7: Reasoning-budget comparison after Phase II RL, averaged over AIME'25, AIME'26, and HMMT'26.

Model	Training	Inference	mean@16	Main tok.	Extension tok.	Total tok.
Qwen3.5-4B	CoT-SFT + GRPO	CoT	63.7	10,338	/	10,338
	CoT-SFT + GRPO*	CoT	64.5	13,121	/	13,121
	ART-SFT + ART-GRPO	**ART**	**66.8**	**9,810**	3,176	12,986
Qwen3.5-9B	CoT-SFT + GRPO	CoT	68.5	10,448	/	10,448
	CoT-SFT + GRPO*	CoT	69.2	12,304	/	12,304
	ART-SFT + ART-GRPO	**ART**	**71.7**	**10,024**	2,098	12,122

* CoT control trained with a longer generation budget, producing total tokens closely matched to ART.

stronger. Moreover, Sec. D shows that the observed extension structure has comparable attention-interaction cost to an equal-length linear trace. Together, these controls suggest that the benefit is not attributable to additional generation alone, but to how demanding local reasoning is organized relative to the main trajectory.

Case study. Fig. 3 illustrates the local reasoning detour targeted by ART. In the matched CoT rollout, a difficult derivation remains on the main trajectory and expands into repeated checks, corrections, and restarts, eventually exhausting the reasoning budget without an answer. In contrast, ART develops two selected local derivations from the current reasoning state in private extensions. Their intermediate traces remain outside the main trajectory, and only the resulting conclusions are later retrieved to continue the solution. This example shows how separating demanding local computation from the main trajectory can prevent a local detour from dominating subsequent reasoning.

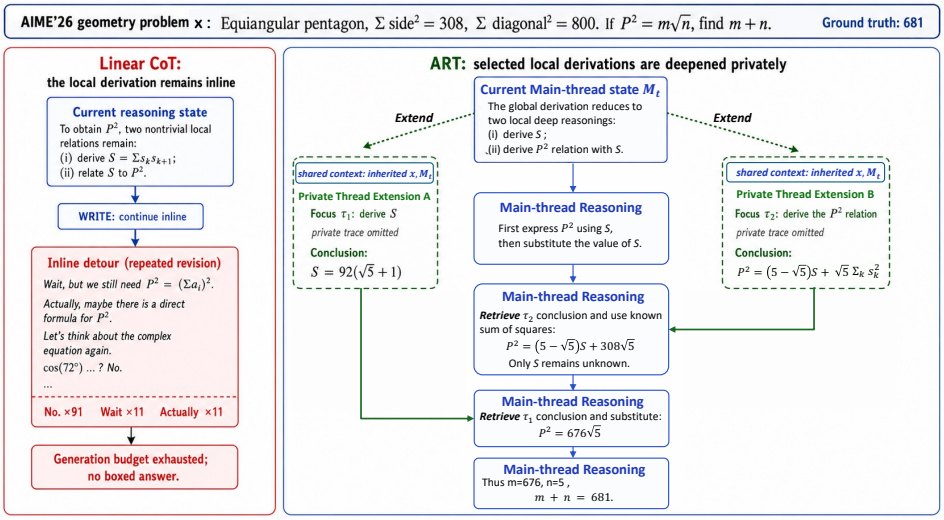

Figure 3: Case study on an AIME'26 problem. Linear CoT falls into a local reasoning detour, while ART develops selected local derivations in private extensions to complete the solution.

5 Conclusion

We studied whether a single LLM can learn to selectively develop intermediate derivations separately and reincorporate their conclusions when needed. To this end, we introduced Adaptive Reasoning Threads (ART), an on-demand, state-driven framework that enables a single LLM to selectively unfold local reasoning from its evolving trajectory. Across model scales and benchmarks, ART consistently improves over matched CoT baselines under comparable reasoning budgets, while its benefits also transfer to ordinary CoT decoding. These results suggest that learning when to extend reasoning on demand is a promising direction for reasoning post-training.

Limitations and future work. Our evaluation focuses on competition mathematics, leaving other reasoning domains for future investigation. A natural next step is to explore ART in settings such as code reasoning and scientific problem solving, where detailed local computations may also support an evolving solution. Across these settings, studying how task characteristics shape the use of local extensions and the incorporation of their conclusions could further inform the design of adaptive reasoning protocols.

AI USE STATEMENT

Generative AI tools were used primarily to assist with writing and language polishing, including improving the clarity, readability, and presentation of author-written text. The research ideas, methodology, experimental design, implementation, analysis, and conclusions were majorly developed by the authors.

REFERENCES

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pages 17682–17690, 2024.

Xingyu Chen, Jiahao Xu, Tian Liang, Zhiwei He, Jianhui Pang, Dian Yu, Linfeng Song, Qiuzhi Liu, Mengfei Zhou, Zhuosheng Zhang, et al. Do not think that much for 2+ 3=? on the overthinking of o1-like llms. *arXiv preprint arXiv:2412.21187*, 2024.

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

Jasper Dekoninck, Nikola Jovanović, Tim Gehrunger, Kári Rögnvaldsson, Ivo Petrov, Chenhao Sun, and Martin Vechev. Beyond benchmarks: Matharena as an evaluation platform for mathematics with llms. 2026. URL https://arxiv.org/abs/2605.00674.

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rstar-math: Small llms can master math reasoning with self-evolved deep thinking. *arXiv preprint arXiv:2501.04519*, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. Deepmath-103k: A large-scale, challenging, decontaminated, and verifiable mathematical dataset for advancing reasoning. 2025. URL https://arxiv.org/abs/2504.11456.

Minki Kang, Wei-Ning Chen, Dongge Han, Huseyin A Inan, Lukas Wutschitz, Yanzhi Chen, Robert Sim, and Saravan Rajmohan. Acon: Optimizing context compression for long-horizon llm agents. *arXiv preprint arXiv:2510.00615*, 2025.

Michael Luo, Sijun Tan, Justin Wong, Xiaoxiang Shi, William Tang, Manan Roongta, Colin Cai, Jeffrey Luo, Tianjun Zhang, Erran Li, Raluca Ada Popa, and Ion Stoica. Deepscaler: Surpassing o1-preview with a 1.5b model by scaling rl, 2025. Notion Blog.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

Jiayi Pan, Xiuyu Li, Long Lian, Charlie Snell, Yifei Zhou, Adam Yala, Trevor Darrell, Kurt Keutzer, and Alane Suhr. Learning adaptive parallel reasoning with language models. *arXiv preprint arXiv:2504.15466*, 2025.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

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

Weiwei Sun, Miao Lu, Zhan Ling, Kang Liu, Xuesong Yao, Yiming Yang, and Jiecao Chen. Scaling long-horizon llm agent via context-folding. *arXiv preprint arXiv:2510.11967*, 2025.

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. Kimi k1. 5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

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

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.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems*, 36:11809–11822, 2023.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2025, 2025.

Yifan Zhang and Team Math-AI. American invitational mathematics examination (aime) 2026, 2026.

Tong Zheng, Hongming Zhang, Wenhao Yu, Xiaoyang Wang, He Xing, Runpeng Dai, Rui Liu, Huiwen Bao, Chengsong Huang, Heng Huang, et al. Parallel-r1: Towards parallel thinking via reinforcement learning. In *International Conference on Learning Representations*, volume 2026, pages 121144–121166, 2026.

Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.

Table 8: Post-training hyperparameters.

Hyperparameter	Value
SFT	
Learning rate	1×10^{-5}
LR schedule	Cosine decay
Final learning rate	1×10^{-6}
Warmup ratio	0.05
Epochs	3
Batch size	16
RL (GRPO)	
Learning rate	1×10^{-6}
Batch size	64
Mini-batch size	64
Rollouts per prompt	8
KL loss coefficient	0.001

A Additional Experimental Details

Data construction and filtering. For supervised cold-start training, we construct a 961-problem set from AIME 1983–2024 and DeepScaleR (Luo et al., 2025). The matched CoT and ART settings use the same problem pool, teacher model, and number of training epochs. Trajectories are retained only when the final answer is correct; ART trajectories are additionally required to contain no protocol errors. For reinforcement learning, both settings use the same 9,117-problem pool filtered from the training sets of DAPO-Math (Yu et al., 2026) and DeepMath (He et al., 2025).

Evaluation metrics and verification. For each problem, we sample $N=16$ independent solutions. Mean@16 is the average accuracy over all sampled solutions, while pass@16 is the fraction of problems solved by at least one sample. Answers are evaluated using a unified automatic verifier that combines MathArena-style normalization (Dekoninck et al., 2026) with symbolic-equivalence checking.

ART-specific metrics. Extension use rate is the fraction of rollouts that successfully launch at least one extension. Protocol-failure rate is computed over extended rollouts and measures the fraction in which at least one launched extension fails to complete a valid ART round trip. Failures include unretrieved extensions, extension or runtime errors, extension truncation, and termination before protocol completion. Extension-retrieval rate is computed at the extension level as the number of successfully retrieved extensions divided by the total number of launched extensions. Extended-rollout accuracy is the accuracy conditioned on successfully launching at least one extension.

Training hyperparameters. Post-training runs in two stages (Sections 3.3.1 and 3.3.2): a supervised cold-start on teacher-distilled ART trajectories, followed by GRPO-style reinforcement learning on a difficulty-filtered math pool. The SFT stage applies the loss only to model-generated spans, while the RL stage optimizes the main-thread policy under a group-normalized advantage with a sequence-level loss aggregation and a small KL penalty against a frozen reference. Tab. 8 lists the settings for both stages.

Inference settings. We use the thinking mode for both training and evaluation. Following prior reasoning-model evaluations (Guo et al., 2025), inference uses a temperature of 0.6, top-p of 0.95, top-k of 20, and a maximum response length of 16,384 tokens.

Reasoning-style metrics. For Tab. 6, all three metrics are corpus-level event frequencies per 10k reasoning characters and use the same eligible traces and denominator. Both the standard-SFT and ART-SFT models are evaluated with direct CoT decoding and no ART tools. We extract the text before `</think>` (or the full response if the closing tag is absent), retain traces of at least 100 characters, sum events and characters within each benchmark, and then macro-average the three benchmark frequencies. Correction frequency counts exact markers "`Wait,`", "`Actually,`",

and "No,"; Wait frequency reports the Wait component separately; and restart frequency counts phrases matching "Let me try/reconsider/redo/rethink."

B TRAINING DYNAMICS DURING RL.



Figure 4: Training dynamics of the 4B ART-GRPO model, showing how local-extension use and extension–retrieval reliability evolve alongside optimization and test performance. Test mean@16 is averaged over AIME'25, AIME'26, and HMMT'26.

Fig. 4 shows how ART behavior evolves throughout ART-GRPO. Extension use rises rapidly, while protocol success and retrieval completion remain high, indicating that the model increasingly exercises local extension as a complete EXTEND–RETRIEVE interaction rather than producing unresolved auxiliary computation. Meanwhile, reward and test accuracy improve as response length decreases, with policy entropy remaining stable. These trends suggest that RL makes local extension an active part of the reasoning process without relying on longer trajectories or degrading interaction reliability.

C ART INTERFACE: PROMPTS AND EXAMPLE TRAJECTORIES

This appendix provides the exact prompts and tool schemas used to serialize ART as a tool-use dialogue, together with representative trajectories for the standard and Python-augmented settings. We retain the assistant's <tool_call>s and the environment's <tool_response>s, while abridging ordinary reasoning inside <think>. The dialogues use the standard chat format (<|im_start|>role ...<|im_end|>), with each environment observation delivered as a user turn containing a <tool_response> block.

C.1 MAIN-THREAD SYSTEM PROMPT

The main thread is conditioned on the prompt below. The two tool schemas are injected into the <tools> block (shown pretty-printed; at runtime each is a single-line JSON object).

Main-thread system prompt

```
You are an expert reasoner. Solve the given problem using ART (Adaptive
    Reasoning Thread) methodology.

# Goal

Produce a solution where:
- The MAIN THREAD is a compact, well-structured solution that
    continuously advances toward the answer.
- Extension inherit main thread reasoning state so far, and handle
    substantial independent sub-tasks whose full derivations would
    clutter the main thread (key lemmas, case analysis, non-trivial
    verification, comparisons of approaches).

# When to extend

Extend ONLY when:
- The sub-task is substantial (not simple arithmetic or an obvious step
    ).
- The main thread does not need the result immediately -- you can
    continue reasoning on other aspects first.
```

```
- Including the full derivation in the main thread would make it longer
     or harder to follow.

Use a private extension when a local subproblem would otherwise
    substantially interrupt the main solution trajectory; otherwise
    continue on the main thread.

# Hard rules

1. Never mix extend and retrieve in the same assistant turn. After
    calling extend, end your response -- the environment will return "
    extension N launched." Then continue main-thread reasoning in the
    next turn. Call retrieve only in a later turn when you genuinely
    need that extension's conclusion. Do not restart the solution or re
    -produce the same extension. If no independent main-thread work
    remains after extending, proceed directly to retrieve in the next
    turn.
2. NEVER restate a extension's derivation in the main thread. When you
    retrieve a extension result, cite only its concise conclusion in
    one sentence and immediately use it to advance the solution.
3. NEVER extend every sub-step. Most reasoning belongs in the main
    thread. A good trace has few, high-value extensions.
4. NEVER extend just to verify simple arithmetic or restate an obvious
    fact.

# Runtime Interaction Protocol

You are interacting with an environment across multiple turns. After
    each turn you output tool_call(s), the environment responds before
    you continue.

- When you call `extend`: write your reasoning first, then end your
    response with the extend tool_call(s). The environment returns "
    Extension N launched." Continue independent main-thread reasoning
    in the next turn. If no independent work remains, call retrieve in
    the next turn instead of restarting or re-extending.
- When you call `retrieve`: end your response with the retrieve
    tool_call(s). The environment returns the extension conclusion. Use
     it to advance the solution in the next turn.
- You may call multiple `extend` tool_calls in the same turn for
    independent extensions. You may call multiple `retrieve` tool_calls
     in the same turn. Never mix extend and retrieve in the same turn.
- Always write reasoning BEFORE tool_call(s). Do not continue writing
    after a tool_call in the same response.

# Quality check

Before finishing, mentally verify:
- Does every extension do work that genuinely does not belong in the
    main thread?
- Did the main thread make substantive progress between extend and
    retrieve across turns?
- Was each retrieve called only when the returned conclusion was
    actually needed to continue? If not, restructure the trace.

# Tools

You may call one or more functions to assist with the user query.

You are provided with function signatures within <tools></tools> XML
    tags:
<tools>
{
```

```
  "type": "function",
  "function": {
    "name": "extend",
    "description": "Spawn a reasoning extension for current reasoning
    state with a specified sub-problem. The extension resolves in the
    local space while the main thread continues its own analysis. Use
    this when you identify a sub-problem that needs deep local
    reasoning.",
    "parameters": {
      "type": "object",
      "properties": {
        "extension_id": {"type": "string", "description": "Sequential
    integer identifier for this extension, starting from 1 and
    incrementing with each new extension (e.g. '1', '2', '3')"},
        "task": {"type": "string", "description": "Clear description of
     the sub-task to solve"}
      },
      "required": ["extension_id", "task"]
    }
  }
}
{
  "type": "function",
  "function": {
    "name": "retrieve",
    "description": "Retrieve the conclusion of a previously launched
    extension. Call this when you need the extension's result to
    continue your reasoning.",
    "parameters": {
      "type": "object",
      "properties": {
        "extension_id": {"type": "string", "description": "The
    extension_id of the extension whose result you need"}
      },
      "required": ["extension_id"]
    }
  }
}
</tools>

For each function call, return a json object with function name and
    arguments within <tool_call></tool_call> XML tags:
<tool_call>
{"name": <function-name>, "arguments": <args-json-object>}
</tool_call>
```

C.2 Extension system prompt

Each extension launches an independent sub-rollout with the system prompt below and a user message assembled from the original problem, the main-thread context at extension time, and the assigned task. The extension must end with a `<conclusion>` block, whose 1–3 sentence content becomes the conclusion z_i returned to the main thread.

Extension system prompt

```
You are a focused reasoning agent assigned to a specific sub-task.

You will receive the original problem, the main thread's context so far
    , and your assigned task.

Rules:
```

```
- Solve only the assigned task. Do not continue the main solution.
- Use the problem and main context only as background -- do not restate
    them.
- Do NOT call any tools.
- Do NOT solve beyond the scope of your task.

When finished, return your result in this format:

<conclusion>
...1-3 sentence result...
</conclusion>
```

The extension user message follows the template:

Extension user message

```
## Original problem
{original problem}

## Main thread context so far
{accumulated main-thread reasoning up to the extension}

## Your task
{task string from the extension call}
```

C.3 Example trajectory (case study)

The trajectory below is the full ART rollout behind our case study: an equiangular-pentagon problem (AIME'26) on which linear CoT loops without reaching an answer, while ART solves it by extending two independent sub-derivations and recombining their conclusions. Notably, the first main-thread turn ends *without* the answer; $676\sqrt{5}$ and $\boxed{681}$ appear only after the two extension conclusions are retrieved. Problem-specific reasoning inside `<think>` and the verbose sub-task/conclusion text are abridged in *gray*; the ART-protocol turns—the assistant's `extend`/`retrieve` calls and the environment returns in green—keep the exact wire format.

ART trajectory: two parallel extensions (equiangular pentagon, AIME'26)

Problem: In an equiangular pentagon, $\sum \text{side}^2 = 308$ and $\sum \text{diagonal}^2 = 800$. Writing $P^2 = m\sqrt{n}$ with n squarefree, find $m + n$. **Answer:** 681

Assistant (main thread) — sets up the problem, then extends two independent sub-derivations in a single turn:

[... places the pentagon on the complex plane with edges $s_k\omega^{k-1}$, notes closure $\sum s_k\omega^{k-1} = 0$, and splits the work into a diagonal-sum step and a Fourier step...]

```
<tool_call>
{"name": "extend", "arguments": {"extension_id": "1", "task":
"derive the adjacent side-product sum S from the diagonal-square
total"}}
</tool_call>
<tool_call>
{"name": "extend", "arguments": {"extension_id": "2", "task":
"via Parseval and the closure relation, express the squared
perimeter through the side-square sum and S"}}
</tool_call>
```

Environment — one acknowledgement per launched extension:

```
<tool_response>
Extension 1 launched.
Continue main-thread reasoning from where you left off.  Do not
```

launch extension 1 again. Call retrieve with extension_id "1" when you need this extension's conclusion.
[active_extensions: 1]
</tool_response>
<tool_response>
Extension 2 launched.
Continue main-thread reasoning from where you left off. Do not launch extension 2 again. Call retrieve with extension_id "2" when you need this extension's conclusion.
[active_extensions: 1, 2]
</tool_response>

Assistant — retrieves both extensions in one turn:

[... both extensions launched; their results are now needed to finish...]

<tool_call>
{"name": "retrieve", "arguments": {"extension_id": "1"}}
</tool_call>
<tool_call>
{"name": "retrieve", "arguments": {"extension_id": "2"}}
</tool_call>

Environment — writes back each extension conclusion z_i:

<tool_response>
[extension_result]
extension_id: 1
task: derive the adjacent side-product sum S from the diagonal-square total
conclusion:
[... from $\sum d_k^2 = 2\sum s_k^2 + 2\cos 72^\circ S \ldots$*]* $S = \sum s_k s_{k+1} = 92(\sqrt{5}+1)$.
[/extension_result]
[active_extensions: 2]
</tool_response>
<tool_response>
[extension_result]
extension_id: 2
task: via Parseval and the closure relation, express the squared perimeter through the side-square sum and S
conclusion:
[... Parseval + closure...] $\left(\sum s_k\right)^2 = (5-\sqrt{5})\,S + \sqrt{5}\sum s_k^2$.
[/extension_result]
[active_extensions:]
</tool_response>

Assistant — substitutes z_1, z_2 and finishes:

[...] $\left(\sum s_k\right)^2 = (5-\sqrt{5})\cdot 92(\sqrt{5}+1) + \sqrt{5}\cdot 308 = 676\sqrt{5}$, so $m = 676,\ n = 5$.
Final answer $\boxed{681}$.

C.4 Extension with Python Tool

We additionally study ART with a Python code_interpreter, introduced during Phase II RL without Python-specific cold-start SFT. We first describe the corresponding interface augmentation and then present a representative rollout illustrating how Python execution is incorporated into extension-local reasoning.

C.4.1 Prompt and Tool-Schema Augmentation

For the Python-tool extension, we modify the prompts above without Python-specific cold-start SFT. On the main thread, we append the following guidance to the system prompt and add a third tool schema to the `<tools>` block after `extend` and `retrieve`. For each extension, we remove the rule "Do NOT call any tools," append the extension-specific guidance below, and expose only the

same code_interpreter schema. Thus, both the main thread and extensions may interleave reasoning with Python execution, while extensions remain unable to extend. Each call runs independently in a sandbox, and its standard output is returned in a <tool_response> block before reasoning continues.

Python-tool prompt additions

```
# Appended to the main-thread system prompt
# When to compute
When a step needs exact calculation, symbolic manipulation, brute-force
     search over a finite space, or numeric verification of a claim,
    call the 'code_interpreter' tool with a short, self-contained
    Python snippet instead of doing it by hand. Print the value you
    need to read, then continue your reasoning from the returned output
    . Keep code minimal; do not offload the whole problem to code.

# Appended to the extension system prompt
You may call the 'code_interpreter' tool for exact computation within
    your assigned sub-task. Do not extend. Print any value you need,
    read the returned output, then give your conclusion.

# Appended tool schema
{
  "type": "function",
  "function": {
    "name": "code_interpreter",
    "description": "Execute Python code in a sandbox and read back its
    stdout. Use this for exact computation, symbolic manipulation,
    brute-force search, or verifying a numeric result, instead of doing
     long arithmetic by hand. Print any value you need to see.",
    "parameters": {
      "type": "object",
      "properties": {
        "code": {
          "type": "string",
          "description": "The Python code to execute. Print the results
     you need to read."
        }
      },
      "required": ["code"]
    }
  }
}
```

C.4.2 Example trajectory with python tool

To illustrate how ART integrates external computation within its extensions, we show a representative successful rollout on an AIME'26 problem. The main thread first reduces the functional condition to a classification of permissible cycle types, then delegates two disjoint groups of types to separate extensions. Each extension uses python-tool to verify its local count and returns only that count: 320 for types containing a 3- or 6-cycle, and 76 for types containing only 1- and 2-cycles. The main thread then combines these partial results to obtain $320 + 76 = \boxed{396}$. Problem-specific reasoning inside <think> is abridged in *gray*; all tool calls, executed Python snippets, tool outputs, and returned conclusions are retained. JSON code strings are line-broken below only for readability.

Python-tool trajectory: extension-local counting (AIME'26)

Problem: Find the number of functions π mapping $A = \{1, 2, 3, 4, 5, 6\}$ onto A such that $\pi^6(a) = a$ for every $a \in A$.

Answer: 396

Assistant (main thread) — classifies the valid cycle types, then extends two disjoint counting cases:

[... surjectivity makes π a permutation; $\pi^6 = \mathrm{id}$ allows only cycle lengths $1, 2, 3, 6$, yielding eight cycle types...]

```
<tool_call>
{"name": "extend", "arguments": {"extension_id": "1", "task":
"Compute the number of permutations of 6 elements with cycle types
containing a 6-cycle, two 3-cycles, a 3-cycle plus a 2-cycle
plus a fixed point, or a 3-cycle plus three fixed points. Use
combinatorial formulas or brute-force enumeration to find the exact
count."}}
</tool_call>
<tool_call>
{"name": "extend", "arguments": {"extension_id": "2", "task":
"Compute the number of permutations of 6 elements with cycle types
consisting only of 1-cycles and 2-cycles: three 2-cycles, two
2-cycles plus two fixed points, one 2-cycle plus four fixed points,
or the identity (six fixed points). Use combinatorial formulas or
brute-force enumeration to find the exact count."}}
</tool_call>
```

Environment — acknowledges both launched extensions:

```
<tool_response>
Extension 1 launched.
[active_extensions: 1]
</tool_response>
<tool_response>
Extension 2 launched.
[active_extensions: 1, 2]
</tool_response>
```

Extension 1 — counts the types containing a 3- or 6-cycle:

[... derives the four cycle-type counts, then verifies them by exhaustive enumeration...]

```
<tool_call>
{"name": "code_interpreter", "arguments": {"code":
"import itertools

def cycle_type(perm):
n = len(perm)
visited = [False]*n
cycles = []
for i in range(n):
if not visited[i]:
cycle = []
j = i
while not visited[j]:
visited[j] = True
cycle.append(j)
j = perm[j]
if len(cycle) > 1:
cycles.append(len(cycle))
return tuple(sorted(cycles))

counts = {None:0}
for p in itertools.permutations(range(6)):
ct = cycle_type(p)
counts[ct] = counts.get(ct, 0) + 1

print(counts)
print('Total:', sum(counts.values()))"}}
</tool_call>
<tool_response>
```

```
{None: 0, (): 1, (2,): 15, (3,): 40, (2, 2): 45, (4,): 90,
(2, 3): 120, (5,): 144, (2, 2, 2): 15, (2, 4): 90, (3, 3): 40,
(6,): 120}
Total: 720
</tool_response>
```

[... reads the enumerated cycle-type counts, selects (6), $(3,3)$, $(2,3)$, and (3), and sums their counts to obtain the assigned subtotal 320...]

```
<conclusion>
The four specified cycle types contribute 120 permutations each for
a single 6-cycle, 40 for two 3-cycles, 120 for a 3-cycle together
with a 2-cycle and a fixed point, and 40 for a single 3-cycle with
three fixed points. Their total is 120 + 40 + 120 + 40 = 320.
</conclusion>
```

Extension 2 — counts the types containing only 1- and 2-cycles:

[... derives the four remaining cycle-type counts, then verifies their sum by exhaustive enumeration...]

```
<tool_call>
{"name": "code_interpreter", "arguments": {"code":
"import itertools

def cycle_type(perm):
n = len(perm)
visited = [False] * n
cycles = []
for i in range(n):
if not visited[i]:
cycle = []
j = i
while not visited[j]:
visited[j] = True
cycle.append(j)
j = perm[j]
if len(cycle) > 0:
cycles.append(len(cycle))
return tuple(sorted(cycles, reverse=True))

valid_count = 0
for p in itertools.permutations(range(6)):
ct = cycle_type(p)
if all(length in (1, 2) for length in ct):
valid_count += 1

print(valid_count)"}}
</tool_call>
<tool_response>
76
</tool_response>
```

[... reads the returned count and confirms that 76 is the subtotal for permutations whose cycles all have length 1 or 2...]

```
<conclusion>
The number of permutations of A with only 1-cycles and 2-cycles is
76.
</conclusion>
```

Assistant (main thread) — retrieves both local conclusions: *[... both extensions have launched; their results are now needed to compute the total...]*

```
<tool_call>
{"name": "retrieve", "arguments": {"extension_id": "1"}}
</tool_call>
<tool_call>
```

{"name": "retrieve", "arguments": {"extension_id": "2"}}
</tool_call>

Environment — returns the two extension conclusions:

<tool_response>
[extension_result]
extension_id: 1
conclusion: The four specified cycle types contribute 120 permutations each for a single 6-cycle, 40 for two 3-cycles, 120 for a 3-cycle together with a 2-cycle and a fixed point, and 40 for a single 3-cycle with three fixed points. Their total is $120 + 40 + 120 + 40 = 320$.
[/extension_result]
[active_extensions: 2]
</tool_response>
<tool_response>
[extension_result]
extension_id: 2
conclusion: The number of permutations of A with only 1-cycles and 2-cycles is 76.
[/extension_result]
[active_extensions:]
</tool_response>

Assistant (main thread) — combines the disjoint local counts and finishes:

[... the first extension covers the four types involving a 3- or 6-cycle, while the second covers the four types using only 1- and 2-cycles; together they exhaust all permissible cycle types...]
The two extensions cover all permissible cycle types and are disjoint. Therefore, the total number of valid permutations is $320 + 76 = 396$. Final answer $\boxed{396}$.

D ATTENTION-INTERACTION ANALYSIS

We analyze the leading quadratic term of causal self-attention for ART relative to a monolithic chain-of-thought (CoT) trace. For a causal sequence of length n, we use $C(n) = \frac{1}{2}n^2$ query–key interactions per layer and head as an analytical proxy for comparing linear and extensioned reasoning.

Analytical model. Consider a problem with prompt length Q. Suppose an ART rollout contains a main-thread generation of M tokens and K extensions of lengths $L_1, \ldots, L_K$, with total extension length $B = \sum_k L_k$. Let $P_k \in [0, M]$ denote the main-thread position at which extension k is launched. A monolithic CoT comparator places the same $M + B$ model-generated reasoning tokens in a single sequence, with attention-interaction cost

$$C_{\text{lin}} = \tfrac{1}{2}(Q + M + B)^2. \tag{9}$$

We assume that each extension reuses the main-thread prefix available at its extending point, attends to no other extension, and returns a summary of length s_k to the main thread. Let $R = \sum_k s_k$ denote the total returned length. Under this prefix-reuse model,

$$C_{ART} = \underbrace{\tfrac{1}{2}(Q + M + R)^2}_{\text{main thread}} + \sum_{k=1}^{K} \underbrace{\left[\tfrac{1}{2}L_k^2 + (Q + P_k)L_k\right]}_{\text{extension } k}. \tag{10}$$

Table 9: Attention-interaction accounting, macro-averaged over AIME 2025, AIME 2026, and HMMT February 2026. The linear comparator uses the same number of model-generated reasoning tokens, $M + B$.

Model	Q	M	B	R	$\bar{P}_{\text{extend}}$	ρ_{macro}
Qwen3.5-4B	1,143	9,810	3,177	842	8,642	1.001
Qwen3.5-9B	1,144	10,024	2,098	376	8,426	0.992

Expanding both costs gives

$$C_{\text{lin}} - C_{ART} = \underbrace{\sum_{i<j} L_i L_j}_{\text{cross-extension isolation}} + \underbrace{\sum_{k=1}^{K} (M - P_k) L_k}_{\text{post-extending context isolation}} - \underbrace{(Q + M)R + \tfrac{1}{2}R^2}_{\text{summary carry-back}}. \tag{11}$$

The first term captures the interactions avoided because independent extensions do not attend to one another. The second captures the main-thread tokens generated after a extension has created, which remain invisible to that extension. These savings are balanced by the cost of carrying returned summaries in the main thread. The relative attention-interaction cost therefore depends on extension lengths, extending positions, and returned-summary length.

For interpretation, we define the extension-length-weighted extending position

$$\bar{P}_{\text{extend}} = \frac{\sum_k P_k L_k}{B}, \tag{12}$$

for $B > 0$, so that the second term in Eq. (11) can be written as $(M - \bar{P}_{\text{extend}})B$.

Empirical accounting. We compute the attention-interaction costs from each rollout before aggregation. For each benchmark d, we sum the rollout-level costs and compute their ratio; we then macro-average over the three benchmarks:

$$\rho_{\text{macro}} = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} \frac{\sum_{r \in d} C_{\text{ART},r}}{\sum_{r \in d} C_{\text{lin},r}}. \tag{13}$$

Here, R includes the non-model-generated protocol tokens appended to the main trace, including tool responses and retrieved-result wrappers. The displayed values of Q, M, B, R, and $\bar{P}_{\text{extend}}$ are reported for interpretation; the reported ρ_{macro} is computed directly from rollout-level costs.

Both ratios are within 1% of one. Thus, under an equal model-generated token budget and the analytical prefix-reuse model, the observed ART extension structure has an attention-interaction cost comparable to that of a linear trace.

Connection to the token-budget analysis. Together with Tab. 7, this analysis indicates that the gains of ART come from organizing computation into extension-local contexts rather than from simply extending a linear trace. At a closely matched generation budget, ART achieves higher accuracy while the observed extension structure retains an attention-interaction cost comparable to that of an equal-length linear trajectory.