

Policy of Thoughts: Scaling LLM Reasoning via Test-time Policy Evolution

Zhengbo Jiao^{1,3*}, Hongyu Xian^{1,2*}, Qinglong Wang^{1,3}, Yunpu Ma⁴,

Zhebo Wang¹, Dezhong Kong^{1,5†} and Meng Han^{1,5}

¹Zhejiang University, ²South China Normal University,

³Shanghai University of Finance and Economics,

⁴LMU Munich, ⁵GenTel.io

* Equal contribution. † Corresponding author.

Abstract

Large language models (LLMs) struggle with complex, long-horizon reasoning due to instability from the frozen-policy assumption. Current *test-time scaling* methods treat execution feedback merely as an external signal for filtering or rewriting trajectories, without internalizing it to improve reasoning strategies. Inspired by Popper’s “conjectures and refutations,” we posit that effective reasoning requires real-time policy adaptation through learning from failures. We introduce **Policy of Thoughts (PoT)**, a framework that recasts reasoning as a within-instance online optimization process. PoT effectively generates diverse candidates via targeted exploration, then applies Group Relative Policy Optimization (GRPO) to update a transient LoRA adapter using execution feedback. This closed-loop design enables dynamic, instance-specific refinement of reasoning priors. Experiments demonstrate PoT’s substantial gains across multiple benchmarks. On LiveCodeBench, a 4B model achieves 49.71% accuracy, surpassing 12 test-time scaling baselines and commercial models (e.g., GPT-4o, DeepSeek-V3) despite being over $50\times$ smaller.

1 Introduction

Recent advances are increasingly positioning large language models (LLMs) as general-purpose reasoners, capable of solving problems that require multi-step deduction, intermediate decisions, and long-horizon planning (Kong et al., 2025). While eliciting explicit intermediate reasoning steps (Chain-of-Thought) (Wei et al., 2022) substantially improves accuracy, reasoning stability remains a central bottleneck on hard instances: these tasks induce a vast search space filled with deceptive trajectories, where a single early mistake can irreversibly derail the entire reasoning chain. We argue that this instability is intrinsic to a frozen policy: without the ability to internalize its own failed attempts, the model lacks a mechanism to

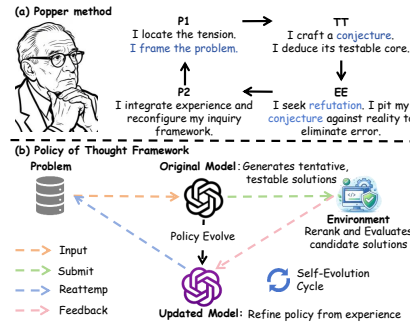


Figure 1: The Policy of Thoughts (PoT) Framework: A Closed-Loop Reasoning System inspired by Popperian epistemology. (a) **The reasoning cycle:** P1 (identify problem), TT (propose conjecture), EE (test against reality), and P2 (update understanding). (b) **PoT implementation:** The model generates solutions (TT); the environment evaluates (EE); feedback is internalized via RL to update the reasoning policy (P2), enabling real-time adaptation.

consistently correct its course and converge to the correct solution.

The mainstream approach to mitigate this problem is *test-time scaling*—increasing inference-time computation (Wang et al., 2025c). Existing methods either expand search width through sampling (Wang et al., 2023; Yao et al., 2023; Kang et al., 2025) or increase depth through iterative refinement (Madaan et al., 2023; Shinn et al., 2023), using execution feedback mainly to filter or revise trajectories. However, they still rely on a frozen policy, spending substantial computation on failed attempts without improving the reasoning strategy itself. In AI history, learning has often surpassed search on hard problems, as shown by AlphaGo (Silver et al., 2017) and AlphaFold (Jumper et al., 2021). This raises a natural question: *can we learn, rather than merely search, at test time?* Recent test-time training (TTT) methods (Akyürek et al., 2025; Zuo et al., 2025; Yuksekgonul et al., 2026) explore this direction via gradient-based updates during inference. However, they target cross-instance adap-

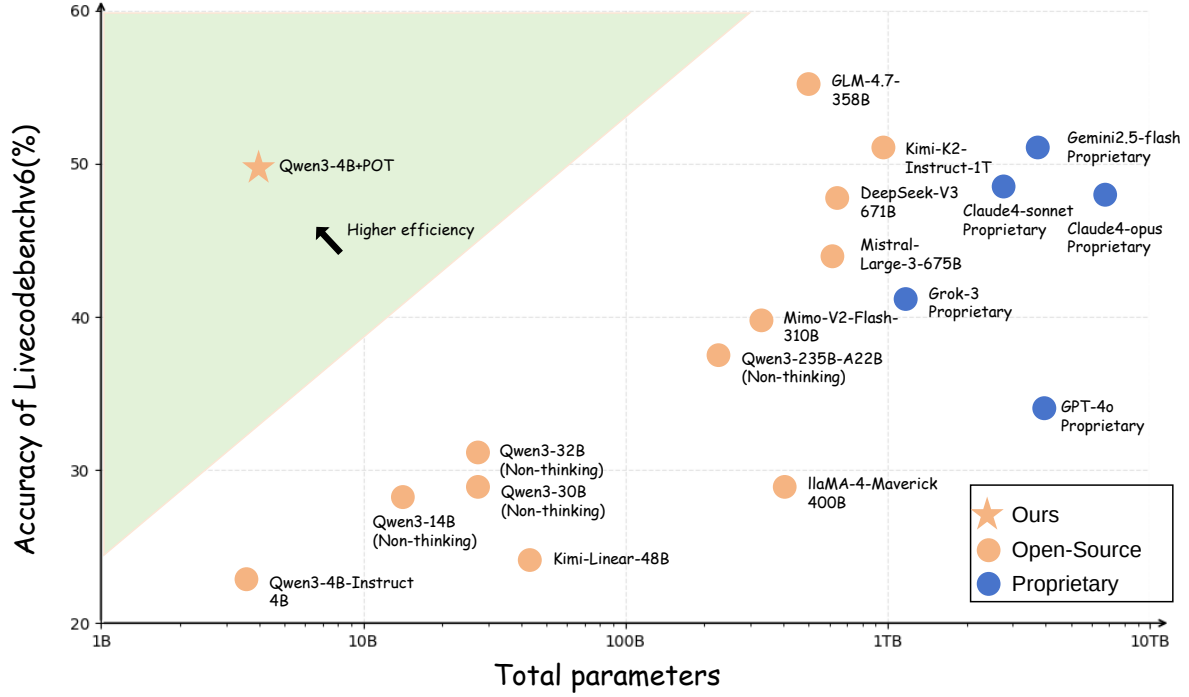


Figure 2: Comparison of model scale versus Live evaluation accuracy. Our PoT-enhanced model achieves 49.71% accuracy, significantly outperforming larger parameter scales—where accuracy degrades with increasing parameters.

tation or long-horizon scientific discovery rather than the per-instance reasoning setting we study.

We introduce **Policy of Thoughts (PoT)**, a framework that effectively recasts test-time reasoning as a within-instance online optimization process. Inspired by the epistemological principle of “conjectures and refutations,” PoT enables the model to iteratively evolve its reasoning policy through a closed loop of exploration and adaptation. Our key design insight is that *per-instance transient adaptation* is the right regime for reasoning problems: unlike standard RL, where the goal is a policy that generalizes across many inputs, each reasoning instance requires its own reasoning strategy, and the adapted policy is merely a means toward finding one correct solution rather than an end in itself (Yuksekgonul et al., 2026). Concretely, PoT first explores diverse candidate solutions via Monte Carlo Tree Search (MCTS), then uses Group Relative Policy Optimization (GRPO) (Shao et al., 2024) to update a transient LoRA adapter based on execution feedback. The adapter is discarded after each problem, leaving the base model untouched. This closed-loop design transforms additional test-time compute into direct policy improvement rather than wasted exploration (see Figure 2). Extensive experiments show that PoT with small models consistently outperforms 16 existing test-time scaling

methods and five commercial large models across multiple benchmarks.

Our contributions are summarized as follows:

1. **Test-time Policy Evolution.** We formalize test-time reasoning as a within-instance online optimization process, shifting the paradigm from static trajectory selection to dynamic policy adaptation. This allows LLMs to transcend the frozen-policy assumption by internalizing execution feedback and reshaping their reasoning strategy in real time.
2. **Policy of Thoughts Framework.** We implement the PoT framework, which unifies structured exploration (MCTS) with parameter-efficient internalization (GRPO + transient LoRA). The per-instance adapter design avoids catastrophic forgetting and cross-instance interference, making PoT well-suited to heterogeneous reasoning tasks.
3. **Superior Empirical Performance.** A PoT-enhanced 4B model achieves 64.02% overall accuracy across five code reasoning benchmarks, surpassing GPT-4o (49.35%), Claude-Opus-4 (55.22%), and Gemini-2.5-Flash (60.91%) despite being over 50× smaller. PoT also improves the reasoning capabilities of thinking models: Qwen3-4B-Thinking accuracy rises from 55.20% to 63.70% on LiveCodeBench V6.

2 Related Work

Test-Time Scaling via Sampling and Refinement.

Chain-of-Thought prompting (Wei et al., 2022) established the basis for scaling test-time computation through intermediate reasoning. Subsequent methods such as Self-Consistency (Wang et al., 2023) and Best-of-N (Kang et al., 2025) improve accuracy by aggregating sampled trajectories. Iterative refinement further enables self-correction: Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) revise outputs with verbal feedback, while CodeT (Chen et al., 2023a) and Self-Debug (Chen et al., 2023b) use execution feedback to verify and repair code. Recent systems such as OpenAI o1 (OpenAI, 2024) further combine extended reasoning with self-correction. Recent work such as Agentic Proposing (Jiao et al., 2026b) further improves reasoning through compositional skill synthesis. However, these methods operate under a *frozen policy*: feedback is used only for filtering, rewriting, or prompting, without updating the underlying reasoning strategy.

Structured Search and Planning. Reasoning can also be formulated as structured search. Tree of Thoughts (Yao et al., 2023) and RAP (Hao et al., 2023) search over candidate thoughts, later extended by LATS (Zhou et al., 2024) and planning-based approaches (Zhang et al., 2023). To improve efficiency, ReST-MCTS (Zhang et al., 2024) and process reward-guided ensembling (Park et al., 2024) incorporate finer-grained feedback into Monte Carlo Tree Search. Recent work including AB-MCTS (Inoue et al., 2025) and RethinkMCTS (Li et al., 2025) further improves search efficiency and thought refinement. SearchEyes (Jiao et al., 2026a) further explores search-world simulation to enhance multimodal deep search intelligence. Despite strong exploration ability, these methods still keep the generation policy fixed throughout search.

Test-Time Training and Adaptation. A growing line of work relaxes the frozen-policy assumption through gradient-based updates at inference. Akyürek et al. (2025) first demonstrated test-time training (TTT) for few-shot learning. This idea has since expanded: TTT-E2E (Tandon et al., 2025) applies TTT to long-context modeling, In-Place TTT (Feng et al., 2026) enables adaptation for Transformer LLMs, LaCT (Zhang et al., 2025) targets efficient 3D reconstruction, and Spatial-TTT (Liu et al., 2026) extends TTT to streaming spatial intelligence. In reasoning, TTRL (Zuo et al., 2025) per-

forms test-time RL with self-consistency rewards, TTT-Discover (Yuksekgonul et al., 2026) applies RL-based TTT to scientific discovery, TEMPO (Zhang et al., 2026) studies scaling through alternating refinement and critic recalibration, and AlphaProof (Hubert et al., 2025) demonstrates test-time RL for formal reasoning. Related work such as Socratic-Zero (Wang et al., 2025a) also explores bootstrapping reasoning through data-free agent co-evolution.

3 Preliminaries

3.1 Monte Carlo Tree Search

MCTS is a decision-making framework designed to explore complex solution spaces by incrementally constructing a search tree. In reasoning tasks, each node represents an intermediate thought or a partial solution state. The search process follows four iterative phases: (1) **Selection**, where nodes are chosen based on the Predictor Upper Confidence Bound applied to Trees (PUCT) to balance exploration and exploitation:

$$a_t = \arg \max_a \left(Q(s, a) + c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_{a'} N(s, a')}}{1 + N(s, a)} \right) \quad (1)$$

where $Q(s, a)$ is the action value, $N(s, a)$ is the visit count, and $P(s, a)$ is the policy prior. (2) **Expansion**, where new potential thoughts are generated by the policy; (3) **Simulation**, where complete reasoning trajectories τ are sampled to reach a terminal state; and (4) **Backpropagation**, where the reward $R(\tau)$ from environment feedback is used to update the values of ancestral nodes:

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{i=1}^{N(s, a)} R(\tau_i) \quad (2)$$

By aggregating terminal outcomes through Equation 2, MCTS filters out erroneous paths and serves as the structured exploration engine for generating diverse candidate trajectories.

3.2 Low-Rank Adaptation

To enable parameter-efficient test-time evolution, we adopt **Low-Rank Adaptation (LoRA)** (Hu et al., 2021). For a frozen pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, LoRA represents the parameter update ΔW as the product of two low-rank matrices

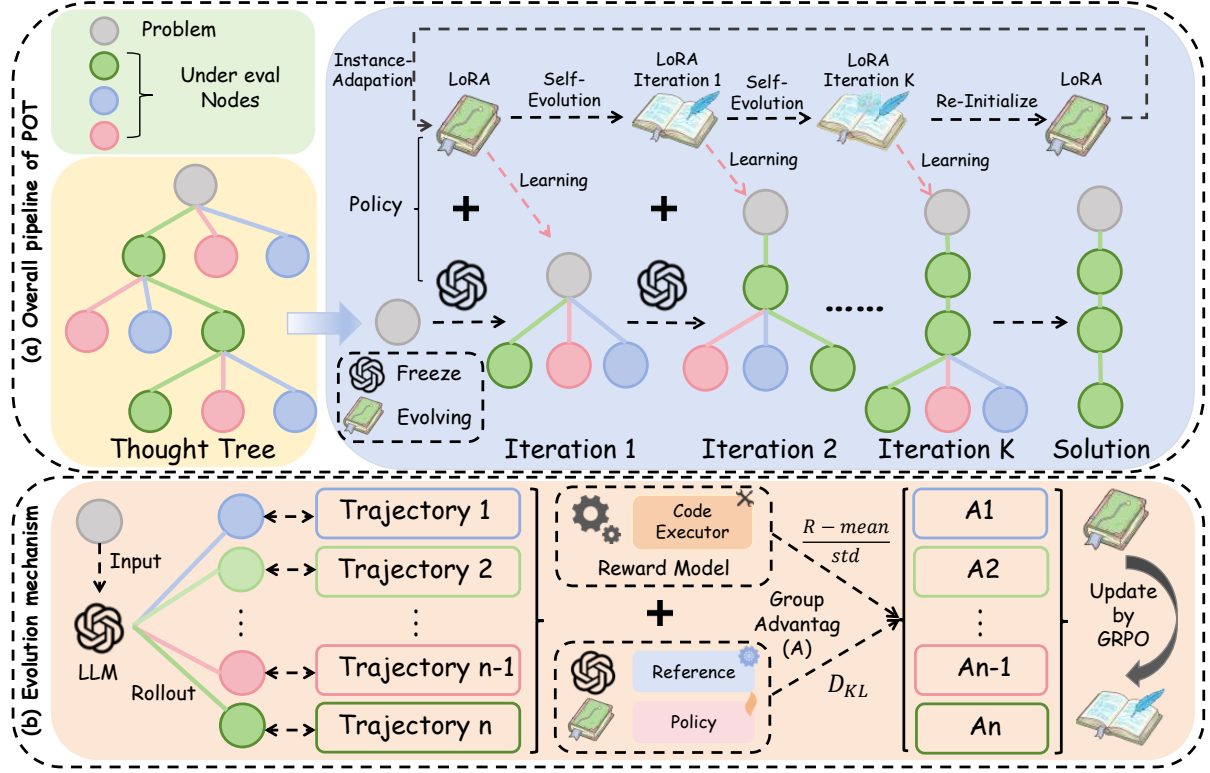


Figure 3: **Policy evolution on a thought tree with frozen LLM and dynamic LoRA adaptation.** (a) **Overall pipeline:** A thought tree is constructed rooted at the problem. The LLM backbone remains frozen while the policy evolves through LoRA adapters. In each iteration, children are expanded from pending nodes, evaluated, and low-quality branches are pruned. (b) **Single iteration mechanism:** All child nodes sharing the same parent form a group. Their trajectories are evaluated and used to compute group-wise advantage signals. The LoRA parameters are then updated via GRPO, enabling structured policy refinement.

$B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, where $r \ll \min(d, k)$. The forward pass is modified as:

$$h = W_0 x + \Delta W x = W_0 x + \frac{\alpha}{r} B A x \quad (3)$$

During a reasoning episode, only A and B are subject to optimization. LoRA acts as a **transient adapter**, providing a lightweight vessel for the rapid parameter mutations needed to transform the initial policy P_1 into the evolved state P_2 .

3.3 Group Relative Policy Optimization

We employ GRPO (Shao et al., 2024) to transform external feedback into internal knowledge. For a given problem q , GRPO samples a group of G trajectories $\{o_1, o_2, \dots, o_G\}$ and optimizes the policy π_θ by maximizing:

$$\mathcal{L}_{\text{GRPO}}(\theta) = \frac{1}{G} \sum_{i=1}^G \left[\min \left(\frac{\pi_\theta(o_i|q)}{\pi_{\text{old}}} \hat{A}_i, \text{clip} \right) - \beta \mathbb{D}_{KL}(\pi_\theta \| \pi_{\text{ref}}) \right] \quad (4)$$

where the advantage $\hat{A}_i$ is estimated via relative rewards within the sampled group:

$$\hat{A}_i = \frac{r_i - \text{mean}(r_1, \dots, r_G)}{\text{std}(r_1, \dots, r_G)} \quad (5)$$

By utilizing group-relative reward normalization, GRPO effectively internalizes environment feedback into the model’s parameters, enabling instance-specific policy adaptation.

4 Methodology: Policy of Thoughts (PoT)

In this section, we present the **Policy of Thoughts (PoT)** framework. We first formalize the reasoning process as an online adaptation problem, followed by an overview of the evolutionary cycle. We then detail the mechanisms of exploratory conjecture and policy (see Figure 3).

4.1 Problem Formulation: Reasoning as Online Adaptation

We formulate problem solving as a **finite-horizon Markov Decision Process (MDP)** $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$.

Unlike static paradigms with a fixed policy, PoT adapts an instance-specific policy $\pi_{\theta,\phi}$ within a single reasoning episode, where θ denotes frozen pre-trained weights and ϕ a **transient adapter**.

- **State Space $\mathcal{S}$:** A state $s_t = (\mathcal{P}, \mathcal{H}_t)$ consists of the problem description $\mathcal{P}$ and the history $\mathcal{H}_t = \{(c_i, f_i)\}_{i=1}^{t-1}$, where c_i is the i -th generated thought and f_i is the corresponding execution feedback.
- **Action Space $\mathcal{A}$:** An action a_t generates a **Thought** c_t , i.e., a functionally complete code implementation as a token sequence $(y_1, \dots, y_L)$, where y_j is the j -th token.
- **Transition $\mathcal{T}$:** Transitions are deterministic: $s_{t+1} = s_t \cup (c_t, f_t)$.
- **Unified Reward $\mathcal{R}$:** Let $\tau = (c_1, c_2, \dots, c_t)$ denote a complete reasoning trajectory—a sequence of thoughts generated for problem $\mathcal{P}$. To quantify the degree of refutation for each conjecture, we define a reward $R(\tau)$ derived from unit test execution:

$$R(\tau) = \begin{cases} 1.0, & \text{if all pass (AC)} \\ N_{\text{pass}}/N_{\text{total}}, & \text{if partially pass} \\ 0, & \text{otherwise} \end{cases} \quad (6)$$

where N_{pass} and N_{total} denote the number of passed and total unit tests, respectively.

4.2 Framework Overview: The Evolution Cycle

Guided by Popperian epistemology, PoT recasts reasoning as a cycle of $P_1 \rightarrow TT \rightarrow EE \rightarrow P_2$ (**Figure 1**). For any given problem $\mathcal{P}$, PoT initializes a task-specific **transient adapter** ϕ_1 , representing the initial policy state P_1 .

The reasoning process unfolds over multiple steps $t \in \{1, \dots, T\}$, each comprising an interleaved phase of exploration and adaptation:

- **Exploratory Conjecture (TT):** PoT employs structured search to explore the solution space, generating a diverse ensemble of trajectories as tentative theories.
- **Policy Internalization ($EE \rightarrow P_2$):** Execution feedback serves as objective refutation. PoT leverages GRPO to **internalize** these signals into the transient adapter, producing an evolved policy ϕ_{t+1} (state P_2) with a refined prior for subsequent search.

This cycle ensures that test-time computation supports both solution-space exploration and internal reasoning refinement.

4.3 Exploratory Conjecture via Structured Search

To generate the diverse conjectures required for policy evolution, PoT performs a search-driven exploration phase. At each step t , it initiates k parallel simulations to populate an experience buffer $\mathcal{B}$, where k is a fixed hyperparameter controlling the breadth of exploration.

Selection and Expansion. The search engine traverses the tree by selecting actions that maximize the PUCT objective:

$$a_t = \operatorname{argmax}_a \left(Q(s, a) + c_{\text{puct}} P(s, a) \frac{\sqrt{\sum N(s, \cdot)}}{1 + N(s, a)} \right) \quad (7)$$

where $N(s, a)$ denotes the visit count, $P(s, a)$ is the prior probability under the current adapter ϕ_t , and $C > 0$ is the exploration constant. Upon reaching a leaf node, k candidate thoughts are generated (k is a fixed expansion width) and evaluated by environment execution.

Backpropagation. The reward $R(\tau)$ is backpropagated to update the value $\hat{Q}$ along the visited path in the tree.

Early Termination. If any simulation yields a perfect reward ($R(\tau) = 1.0$), the search terminates immediately and returns the corresponding solution. Otherwise, the ensemble of G trajectories $\mathcal{B} = \{(\tau_i, R(\tau_i))\}_{i=1}^G$ is forwarded to the internalization phase, where G denotes the group size (i.e., the number of collected trajectories per iteration).

4.4 Policy Internalization via Online Optimization

When exploration fails to yield a solution, PoT converts feedback from failed conjectures into policy-level updates, constituting the core of **test-time policy evolution**.

Relative Advantage Estimation. To internalize refutation signals, we compute the relative advantage $\hat{A}_i$ within the group $\mathcal{B}$. This identifies comparatively more promising conjectures and provides the gradient direction for policy evolution:

$$\hat{A}_i = \operatorname{clip} \left(\frac{R(\tau_i) - \mu_G}{\max(\sigma_G, \eta)}, -C_A, C_A \right) \quad (8)$$

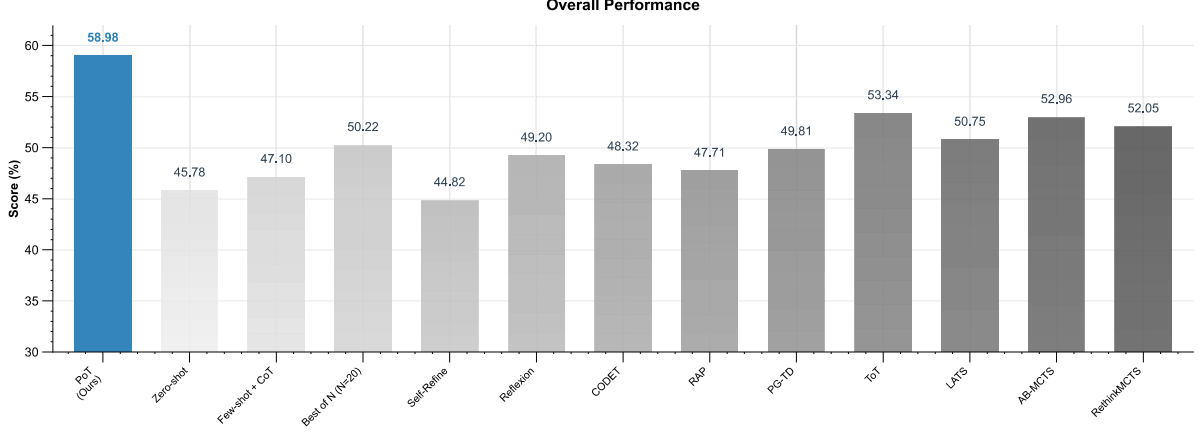


Figure 4: **Overall performance on reasoning benchmarks.** PoT achieves the highest score (58.98%), significantly outperforming self-refinement methods, search-based approaches, and standard inference baselines. It also surpasses recent MCTS-enhanced methods such as AB-MCTS and ReinforceMCTS, demonstrating the effectiveness of its unified thought-tree-guided policy evolution framework.

where $\mu_G = \text{mean}(\{R_j\}_{j=1}^G)$ and $\sigma_G = \text{std}(\{R_j\}_{j=1}^G)$. where $\eta > 0$ is a small constant to prevent division by zero, $C_A > 0$ is the clipping bound for advantage normalization, and $\text{mean}(\cdot)$, $\text{std}(\cdot)$ denote the empirical mean and standard deviation over the group.

Gradient-based Internalization. We minimize the GRPO loss to update the transient adapter ϕ , directly encoding refutation signals into the model’s parameters. Let $r_{i,j}(\phi) = \pi_\phi(y_{i,j} | s, y_{i,<j}) / \pi_{\phi_t}(y_{i,j} | s, y_{i,<j})$ denote the importance sampling ratio. The loss is:

$$\mathcal{L}(\phi) = \frac{1}{G} \sum_{i=1}^G \frac{1}{|\tau_i|} \sum_{j=1}^{|\tau_i|} \left[\min(r_{i,j} \hat{A}_i, \text{clip}(r_{i,j}, 1-\epsilon, 1+\epsilon) \hat{A}_i) - \beta \mathbb{D}_{\text{KL}}(\pi_\phi \| \pi_{\text{ref}}) \right] \quad (9)$$

Here, $|\tau_i|$ denotes the token count of trajectory τ_i , $y_{i,<j}$ is the token prefix up to position $j-1$, $\epsilon > 0$ is the PPO clipping threshold, $\beta \geq 0$ controls KL regularization strength, and π_{ref} is the reference policy (typically ϕ_t).

Action Commitment. After E epochs of optimization (where E is a fixed number of fine-tuning steps), the model commits to a new thought c_t via greedy decoding from the evolved policy $\pi_{\theta, \phi_{t+1}}$. This updated policy then serves as the prior for the next step’s exploration, progressively narrowing the search toward high-quality regions. Upon episode completion, the transient adapter is discarded, resetting the model for the next independent task.

5 Experiments

5.1 Experiment Setup

Models. We used Qwen3-4B-Instruct-2507 (Yang et al., 2025), Qwen3-1.7B-Thinking (Yang et al., 2025), Qwen3-30B-A3B-Instruct-2507 (Yang et al., 2025), and Phi-4-mini-4k-instruct (Microsoft, 2025) as the base models on which we deploy PoT and SOTA baseline methods.

Benchmarks. We evaluated our method on five benchmarks: LiveCodeBench v5 and v6 (Jain et al., 2024), HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and the ICPC subset of OJBench (Wang et al., 2025b). The complete benchmark details are provided in Section F, which includes the specific number of problems in each dataset: HumanEval (164 problems), MBPP (257 problems), LiveCodeBench v5 lite (167 problems), LiveCodeBench v6 lite (175 problems), and ICPC (73 problems), totaling 836 programming problems spanning various difficulty levels.

Baselines. We evaluate PoT against a comprehensive suite of baselines spanning prompting, iterative refinement, structured search, test-time training, and frontier LLMs. Detailed configurations, hyperparameters, and fairness protocols for all baselines are provided in Appendix B.

Evaluation Protocol. We evaluate all methods using pass@1. To ensure fair comparison, PoT is matched with search-based and TTT baselines under a consistent computational budget; detailed configurations and evaluation protocols are provided in Appendix D.

Implementation Details. See in Appendix E.

Table 1: Code reasoning results with a 4B solver. All methods use the same inference budget and evaluation protocol (Appendix B). LiveCodeBench (LCB) results are evaluated on the code_generation_lite split. Overall is the unweighted average across all five benchmarks.

Baselines	HumanEval	MBPP	LCB V5	LCB V6	ICPC	Overall
Model Baselines (Strong Commercial models (zero-shot))						
Qwen3-235B-A22B	79.88	75.10	35.33	37.21	5.48	46.60
DeepSeek-V3	91.50	87.60	31.74	16.00	9.59	47.29
Gemini-2.5-Flash	94.01	81.70	64.07	56.57	8.22	60.91
GPT-4o	92.70	87.60	29.94	29.71	6.80	49.35
Claude-Opus-4	95.80	83.20	50.30	40.00	6.80	55.22
Method Baselines (On Qwen3-4B-Instruct-2507)						
<i>Standard Inference</i>						
+Zero-shot	84.76	79.77	30.54	27.43	6.80	45.86
+Few-shot + CoT	85.98	82.88	31.13	28.57	8.22	47.36
+Best of N (N=20)	90.85	82.10	43.61	37.71	9.59	52.77
<i>Self-Refinement / Debugging</i>						
+Self-Refine	84.15	80.13	30.54	28.00	5.48	45.66
+Reflexion	87.80	85.60	34.13	38.12	10.59	51.25
+CODET	88.41	83.79	34.43	30.13	8.22	49.00
+RAP	87.19	81.71	31.73	28.00	8.22	47.37
<i>Search-Based Reasoning</i>						
+PG-TD	89.02	83.72	38.12	34.32	10.59	51.15
+ToT	90.24	82.37	43.11	38.86	16.43	54.20
+LATS	87.80	90.66	47.51	43.43	13.69	56.62
+AB-MCTS	90.85	83.66	39.52	36.00	15.07	53.02
+RethinkMCTS	89.02	87.34	49.83	44.12	15.07	57.08
<i>Test-Time Training / Adaptation</i>						
+TTRL	91.22	86.14	48.61	41.67	13.70	56.27
+TTT-Discover	93.90	88.76	51.38	44.86	15.07	58.79
+RSA	88.78	86.91	48.94	43.71	14.85	56.64
+s1	90.37	88.02	50.11	45.03	15.66	57.84
+PoT (Ours)	98.78	94.94	57.49	49.71	19.18	64.02

Table 2: Cross-architecture generalization results. PoT is applied to models of different families and scales, including instruct variants. LCB results use the code_generation_lite split. Overall is the unweighted average across evaluated benchmarks.

Method	HumanEval	MBPP	LCB V5	LCB V6	ICPC	Overall
Phi-4-mini-4k-instruct						
+zero-shot	74.39	68.09	19.90	15.42	–	44.45
+PoT (Ours)	87.80	82.49	33.53	27.42	–	57.81
Qwen3-1.7B-Thinking						
+zero-shot	78.05	70.82	32.93	29.14	4.11	43.01
+PoT (Ours)	90.24	84.05	45.51	42.29	8.23	54.06
Qwen3-30B-A3B-Instruct-2507						
+zero-shot	93.30	85.20	65.00	40.20	25.97	61.93
+PoT (Ours)	98.78	96.89	82.04	62.29	41.10	76.22

5.2 Main Results

As shown in Table 1, PoT achieves an overall score of 64.02 on Qwen3-4B-Instruct-2507, outperforming the strongest prompting baseline (Best-of-20, 52.77) by +11.25 points, the best search method (RethinkMCTS, 57.08) by +6.94 points, and the best test-time training method (TTT-Discover, 58.79) by +5.23 points. Notably, PoT surpasses commercial models including GPT-4o (49.35), Claude-Opus-4 (55.22), and Gemini-2.5-

Flash (60.91) despite being over $50\times$ smaller. The largest gains appear on challenging benchmarks: +7.66 on LCB V5 and +5.59 on LCB V6, demonstrating that test-time policy evolution is particularly effective for complex reasoning tasks requiring iterative refinement.

5.3 Cross-Architecture Generalization

Table 2 shows that PoT generalizes across model families, scales, and training paradigms. On Qwen3-1.7B-Thinking, PoT improves overall accu-

Table 3: Ablation on test-time evolution using Qwen3-4B-Instruct-2507 on LCB v6. Δ denotes absolute percentage point gain relative to the zero-shot baseline.

Configuration	Accuracy (%)	Δ (%)
<i>Zero-shot</i>		
Base Model	27.43	–
<i>Search Only (No Adaptation)</i>		
PoT w/o LoRA Update	37.14	$\uparrow 9.71$
<i>Search + Adaptation (Ours)</i>		
PoT w/ LoRA Update	49.71	$\uparrow 22.28$

Table 4: Ablation on branching factor k on LCB v6. Cost denotes measured wall-clock latency per evolution step. Δ is relative to $k = 3$ (ours).

Branching Factor k	Accuracy (%)	Δ (%)	Cost (ms)
$k = 3$ (Ours)	49.71	–	473.66
$k = 1$ (Greedy)	31.42	$\downarrow 18.29$	161.43
$k = 2$	41.14	$\downarrow 8.57$	309.87
$k = 4$	50.28	$\uparrow 0.57$	849.47
$k = 8$	54.86	$\uparrow 5.15$	2369.96
$k = 16$	55.42	$\uparrow 5.71$	3984.52
$k = 32$	56.57	$\uparrow 6.86$	8752.96

Table 5: Ablation on LoRA rank r , learning rate η on LCB v6. Δ denotes absolute difference relative to ($r = 8, \eta = 10^{-4}$) (ours).

Setting (r, η)	Accuracy (%)	Δ (%)
LoRA(8, 1×10^{-4}) (Ours)	49.71	–
LoRA(4, 5×10^{-5})	43.43	$\downarrow 6.28$
LoRA(4, 1×10^{-4})	46.29	$\downarrow 3.42$
LoRA(8, 5×10^{-5})	46.86	$\downarrow 2.85$
LoRA(16, 5×10^{-5})	50.28	$\uparrow 0.57$
LoRA(16, 1×10^{-4})	50.86	$\uparrow 1.15$

racy from 43.01 to 54.06 (+11.05 points), while Phi-4-mini-4k-instruct gains +13.36 points. The trend also holds for larger models, with Qwen3-30B-A3B-Instruct-2507 improving by +14.29 points. PoT is also complementary to thinking models; additional results are in Appendix H. These results confirm that test-time policy evolution is architecture-agnostic and enhances long-form reasoning.

5.4 Ablation Study

All ablations use Qwen3-4B-Instruct-2507 on LiveCodeBench v6.

Ablation on Test-time Evolution. Table 3 demonstrates the critical importance of dynamic weight adaptation in PoT. Starting from the Qwen3-4B zero-shot baseline (27.43%), the version without LoRA updates reaches only 37.14%

through pure MCTS exploration. In contrast, our full framework achieves 49.71%—a 12.57-point improvement over static search. This margin highlights how on-the-fly internalization of execution feedback enables the model to escape logical plateaus that often limit traditional search, providing essential adaptability for more successful reasoning.

Ablation on Branching Factor k . Table 4 investigates the impact of search breadth by varying k . Results show $k = 3$ optimally balances exploration diversity and computational efficiency. Lower values ($k = 1, 2$) yield significantly worse performance—e.g., $k = 1$ drops to 31.42% due to insufficient contrastive samples for GRPO—while large k incurs prohibitive overhead. Although $k = 16$ achieves 55.42%, its latency (3984.52 ms) is over eight times that of $k = 3$ (473.66 ms), indicating severely diminished returns. Thus, $k = 3$ provides the ideal equilibrium, delivering robust performance while maintaining high iteration speed for effective test-time evolution.

Ablation on LoRA Hyperparameters. Table 5 identifies the stability-performance frontier through joint analysis of rank r and learning rate η . The ($r = 8, \eta = 10^{-4}$) configuration achieves the highest accuracy (49.71%), outperforming others by 2.28–6.28%. Reducing either r or η leads to drops up to 6.28%, suggesting insufficient capacity for immediate knowledge internalization. Increasing r to 16 yields marginal gains but risks overfitting on sparse per-instance feedback.

5.5 Reward Robustness and Cross-Domain Generality

Sensitivity to Reward Quality. A key question is whether PoT benefits specifically from meaningful execution feedback, or merely from the additional computation introduced by online adaptation. To isolate the effect of the learning signal, we keep the MCTS search procedure unchanged and perturb only the reward used for GRPO updates. Results on LCB V6 with Qwen3-4B are shown in Table 6. Incorrect reward signals consistently degrade performance below the search-only baseline (37.14%), while progressively reducing reward noise leads to steadily improved performance. In particular, reversing the reward direction causes performance to fall to 25.14%, even below zero-shot generation (27.43%). This demonstrates that the LoRA adapter does not provide a generic benefit from additional computation; rather, it internalizes the

GRPO Reward Setting	Pass@1 (%)
Inverted ($1 - R$)	25.14
Zero-shot	27.43
Shuffled	31.43
Noisy ($\sigma = 1.0$)	33.71
Constant	36.00
Search only (no LoRA)	37.14
Noisy ($\sigma = 0.7$)	39.43
Noisy ($\sigma = 0.5$)	43.43
Noisy ($\sigma = 0.3$)	46.29
Noisy ($\sigma = 0.1$)	48.57
PoT (full)	49.71

Table 6: Sensitivity of PoT to the quality of the GRPO reward signal on LCB V6. MCTS search is kept identical across all settings.

direction encoded by the reward signal. With Gaussian noise, performance improves monotonically from 33.71% at $\sigma = 1.0$ to 48.57% at $\sigma = 0.1$, approaching the full PoT result of 49.71%. These results indicate that reward fidelity is an important determinant of the effectiveness of test-time policy evolution.

Generalization Beyond Code Generation. We further evaluate whether policy evolution is specific to code generation or can be applied to tasks with substantially different output structures and feedback mechanisms. Table 7 summarizes results across environment-verified and reward-model-based tasks. PoT consistently improves over zero-shot generation across code generation, formal theorem proving, mathematical reasoning, knowledge question answering, and multimodal reasoning. Notably, PutnamBench and miniF2F provide a particularly strong test of generality: formal proof checking supplies only binary feedback and proofs are not naturally compositional in the same manner as code. Nevertheless, PoT improves DeepSeek-Prover-V2-7B from 9.00% to 30.20% on PutnamBench and from 48.57% to 65.14% on miniF2F. These results suggest that composability is not a strict prerequisite for policy evolution; instead, sufficiently informative variation in the reward signal can provide a useful learning signal for transient adaptation.

Adaptation Dynamics. We further inspect individual adaptation trajectories to understand how PoT improves reasoning within a problem. On LCB V6 problem #127, the model initially adopts a greedy strategy and achieves a pass rate of 3/14. After successive policy updates, it shifts to backtracking (4/14), introduces memoization (8/14), and fur-

Task	Model	Zero-shot	PoT
LCB V6	Qwen3-4B	27.43	49.71
PutnamBench	DeepSeek-Prover-V2-7B	9.00	30.20
miniF2F	DeepSeek-Prover-V2-7B	48.57	65.14
TriviaQA	Qwen2.5-7B	59.70	75.00
AIME-25	Qwen3-8B	20.90	32.30
MMLU-Pro	Qwen3-8B	63.40	74.10
MathVista	Qwen3-VL-8B	53.90	65.40

Table 7: Cross-domain evaluation of PoT. The method consistently improves performance across environment-verified and reward-model-based tasks.

ther improves the implementation with prefix-sum optimization and boundary corrections, eventually reaching 14/14. In contrast, on problem #43, the model moves from brute-force enumeration toward algebraic constructions, but remains at 0–2/18 after 20 iterations. The latter failure illustrates an important limitation: when candidate solutions receive similarly sparse rewards, the learning signal may be insufficient to distinguish promising mathematical directions. These trajectories suggest that PoT is most effective when execution feedback provides sufficient reward variation to guide both strategy transitions and subsequent refinement.

6 Conclusion

We present Policy of Thoughts (PoT), a test-time reasoning framework that moves beyond the frozen-policy assumption by enabling per-instance online policy evolution. PoT integrates structured exploration (MCTS) with gradient-based internalization (GRPO + transient LoRA), allowing the model to dynamically refine its reasoning strategy by learning from failed attempts rather than discarding them. Across five code reasoning benchmarks, a PoT-enhanced 4B model achieves 64.02% overall accuracy—surpassing 16 test-time scaling baselines including recent TTT methods (TTT-Discover, TTRL), as well as commercial models such as GPT-4o (49.35%), Claude-Opus-4 (55.22%), and Gemini-2.5-Flash (60.91%), despite being over $50\times$ smaller. PoT is also complementary to thinking models, further boosting Qwen3-4B-Thinking from 55.20% to 63.70% on LiveCodeBench V6. Ablations confirm that the core gain stems from coupling adaptation with search: pure exploration yields only marginal improvement, while the per-instance transient adapter design outperforms persistent cross-instance adaptation by 5.41 points.

Limitations

We acknowledge that the reliance on learned reward models as proxies for ground-truth feedback introduces a degree of subjectivity. Whether model-based signals can fully substitute for deterministic environmental feedback remains an open question and represents an interesting direction for future investigation.

Acknowledgement

This work is supported by the “Pioneer” and “Leading Goose” R&D Program of Zhejiang (Grant No. 2026SDXT012).

References

- Ekin Akyürek, Mehul Damani, Linlu Qiu, Han Guo, Yoon Kim, and Jacob Andreas. 2025. The surprising effectiveness of test-time training for abstract reasoning. In *Proceedings of the 42nd International Conference on Machine Learning*.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. [Program synthesis with large language models](#). *Preprint*, arXiv:2108.07732.
- Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023a. [Codet: Code generation with generated tests](#). In *International Conference on Learning Representations*.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, and Scott Gray et al. 2021. [Evaluating large language models trained on code](#). *Preprint*, arXiv:2107.03374.
- Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023b. [Teaching large language models to self-debug](#). *Preprint*, arXiv:2304.05128.
- Guhao Feng, Shengjie Luo, Kai Hua, Ge Zhang, Di He, Wenhao Huang, and Tianle Cai. 2026. [In-place test-time training](#). *Preprint*, arXiv:2604.06169.
- Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. 2023. [Reasoning with language model is planning with world model](#). *Preprint*, arXiv:2305.14992v2.
- Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. [Lora: Low-rank adaptation of large language models](#). *Preprint*, arXiv:2106.09685.
- Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, and Ottavia Bertolli et al. 2025. [Olympiad-level formal mathematical reasoning with reinforcement learning](#). *Nature*, 651(8106):607–613.
- Yuichi Inoue, Kou Misaki, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. 2025. [Wider or deeper? scaling llm inference-time compute with adaptive branching tree search](#). In *Advances in Neural Information Processing Systems*.
- Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. [Live-codebench: Holistic and contamination free evaluation of large language models for code](#). *Preprint*, arXiv:2403.07974.
- Zhengbo Jiao, Yiming Cheng, Yilei Jiang, Kaituo Feng, Rui Huang, Tianyi Jiang, Juanxi Tian, Qunzhong Wang, Tailai Chen, and Qianshan Wei et al. 2026a. [Searcheyes: Towards frontier multimodal deep search intelligence via search world simulation](#). *arXiv preprint arXiv:2607.05943*.
- Zhengbo Jiao, Shaobo Wang, Zifan Zhang, Xuan Ren, Wei Wang, Bing Zhao, Hu Wei, and Linfeng Zhang. 2026b. [Agentic proposing: Enhancing large language model reasoning via compositional skill synthesis](#). *arXiv preprint arXiv:2602.03279*.
- John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, and Anna Potapenko et al. 2021. Highly accurate protein structure prediction with alphafold. *Nature*, 596(7873):583–589.
- Zhewei Kang, Xuandong Zhao, and Dawn Song. 2025. [Scalable best-of-n selection for large language models via self-certainty](#). *Preprint*, arXiv:2502.18581.
- Dezhang Kong, Shi Lin, Zhenhua Xu, Zhebo Wang, Minghao Li, Yufeng Li, Yilun Zhang, Hujin Peng, Xiang Chen, and Zeyang Sha et al. 2025. A survey of llm-driven ai agent communication: Protocols, security risks, and defense countermeasures. *arXiv preprint arXiv:2506.19676*.
- Qingyao Li, Wei Xia, Xinyi Dai, Kounianhua Du, Weiqun Liu, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025. [Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 8092–8110. Association for Computational Linguistics.
- Fangfu Liu, Diankun Wu, Jiawei Chi, Yimo Cai, Yi-Hsin Hung, Xumin Yu, Hao Li, Han Hu, Yongming Rao, and Yueqi Duan. 2026. [Spatial-ttt: Streaming visual-based spatial intelligence with test-time training](#). *Preprint*, arXiv:2603.12255.

- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, and Yiming Yang et al. 2023. [Self-refine: Iterative refinement with self-feedback](#). *Preprint*, arXiv:2303.17651.
- Microsoft. 2025. [Phi-4-Mini Technical Report: Compact yet Powerful Multimodal Language Models via Mixture-of-LoRAs](#). *Preprint*, arXiv:2503.01743.
- OpenAI. 2024. [Openai o1 system card](#). *Preprint*, arXiv:2412.16720.
- Sungjin Park, Xiao Liu, Yeyun Gong, and Edward Choi. 2024. [Ensembling large language models with process reward-guided tree search for better complex reasoning](#). *Preprint*, arXiv:2412.15797.
- Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. 2024. [Deepseekmath: Pushing the limits of mathematical reasoning in open language models](#). *Preprint*, arXiv:2402.03300.
- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. [Reflexion: Language agents with verbal reinforcement learning](#). *Preprint*, arXiv:2303.11366.
- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, and Adrian Bolton et al. 2017. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359.
- Arnub Tandon, Karan Dalal, Xinhao Li, Daniel Kocreja, Marcel Rød, Sam Buchanan, Xiaolong Wang, Jure Leskovec, Sanmi Koyejo, Tatsunori Hashimoto, Carlos Guestrin, Jed McCaleb, Yejin Choi, and Yu Sun. 2025. [End-to-end test-time training for long context](#). *Preprint*, arXiv:2512.23675.
- Shaobo Wang, Zhengbo Jiao, Zifan Zhang, Yilang Peng, Xu Ze, Boyu Yang, Wei Wang, Hu Wei, and Linfeng Zhang. 2025a. Socratic-zero: Bootstrapping reasoning via data-free agent co-evolution. *arXiv preprint arXiv:2509.24726*.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. [Self-consistency improves chain of thought reasoning in language models](#). *Preprint*, arXiv:2203.11171.
- Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. 2025b. [Ojbench: A competition level code benchmark for large language models](#). *Preprint*, arXiv:2506.16395.
- Ziqi Wang, Boye Niu, Zipeng Gao, Zhi Zheng, Tong Xu, Linghui Meng, Zhongli Li, Jing Liu, Yilong Chen, Chen Zhu, Hua Wu, Haifeng Wang, and En-hong Chen. 2025c. [A survey on parallel reasoning](#). *Preprint*, arXiv:2510.12164.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.
- An Yang, Anfeng Li, Baosong Yang, and Beichen Zhang et al. 2025. [Qwen3 technical report](#). *Preprint*, arXiv:2505.09388.
- Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. [Tree of thoughts: Deliberate problem solving with large language models](#). *Preprint*, arXiv:2305.10601.
- Mert Yuksekgonul, Daniel Kocreja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. 2026. [Learning to discover at test time](#). *Preprint*, arXiv:2601.16175.
- Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. [Rest-mcts*: Llm self-training via process reward guided tree search](#). In *Advances in Neural Information Processing Systems*.
- Qingyang Zhang, Xinke Kong, Haitao Wu, Qinghua Hu, Minghao Wu, Baosong Yang, Yu Cheng, Yun Luo, Ganqu Cui, and Changqing Zhang. 2026. [Tempo: Scaling test-time training for large reasoning models](#). *Preprint*, arXiv:2604.19295.
- Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. [Planning with large language models for code generation](#). In *International Conference on Learning Representations (ICLR)*.
- Tianyuan Zhang, Sai Bi, Yicong Hong, Kai Zhang, Fujun Luan, Songlin Yang, Kalyan Sunkavalli, William T. Freeman, and Zexiang Xu. 2025. [Test-time training done right](#). *Preprint*, arXiv:2505.23884.
- Li Zhong, Zilong Wang, and Jingbo Shang. 2024. [Debug like a human: A large language model debugger via verifying runtime execution step by step](#). *Preprint*, arXiv:2402.16906v6.
- Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. 2024. [Language agent tree search unifies reasoning acting and planning in language models](#). *Preprint*, arXiv:2310.04406.
- Yuxin Zuo, Kaiyan Zhang, Shang Qu, Li Sheng, Xuekai Zhu, Biqing Qi, Youbang Sun, Ganqu Cui, Ning Ding, and Bowen Zhou. 2025. [Ttrl: Test-time reinforcement learning](#). *Preprint*, arXiv:2504.16084.

A Generalization and Future Directions

To assess the generality of PoT beyond code generation, we evaluate it across diverse reasoning domains spanning theorem proving, mathematical reasoning, knowledge-intensive QA, and multimodal understanding.

A clear trend emerges across these tasks. PoT exhibits the strongest gains in settings with *native environment feedback*, such as formal proof checkers or tool-execution environments, where refutation signals are precise, deterministic, and directly actionable. In contrast, for tasks without executable interfaces (e.g., open-ended math or knowledge QA), we employ RRM-7B as a unified verifier to score candidate solutions. While performance in these settings is naturally bounded by verifier reliability, PoT still achieves consistent improvements.

We evaluate PoT across five representative benchmarks using reported baseline scores and PoT-enhanced results. As shown in Table 8, the largest improvements occur in PutnamBench and DeepResearch—both of which provide full environment feedback—whereas RM-based tasks show more moderate but stable gains.

These results demonstrate PoT’s broad effectiveness across heterogeneous reasoning tasks, while highlighting reliable reasoning verifiers as a promising future direction.

Ablation on Reward Signal Quality. To disentangle the impact of verifier capability on PoT’s performance in non-executable domains, we conduct an ablation study on AIME-25 using different reward sources (Table 9). We compare the standard setup against *Self-as-RM* (where the policy model evaluates its own outputs) and stronger proprietary or open-source reasoning RMs (RRM-32B, Gemini-3-Pro, GPT-5).

The results reveal a clear hierarchy: Reasoning RMs are effective surrogates: Specialized verifiers like RRM-7B/32B bridge the gap significantly, and scaling to frontier models (Gemini-3-Pro, GPT-5) further amplifies performance. This indicates that while *environment-dense* tasks (like coding) naturally favor PoT due to deterministic feedback, the framework is robust enough to leverage high-quality *reasoning-based* feedback to achieve comparable trajectories in abstract domains.

These findings underscore a key insight: PoT is best suited for *agentic, grounded reasoning*. However, as reward models evolve from simple classifiers to complex reasoners, the distinction between

"environment feedback" and "RM feedback" begins to blur, allowing PoT to extend its efficacy into purely conceptual domains. Future work will focus on narrowing the gap between open-weight RMs and ground-truth environments for scientific discovery.

B Baseline Setup

We unify the execution environment for all baselines to ensure consistent comparison. All baselines run under identical inference hardware and software configurations, with shared backend, timeout settings, and dataset processing pipelines.

Simple (0-shot). Temperature is set to 0.0 with greedy decoding. A single code solution is generated without iterative refinement. The maximum output length per generation is set to 8192 tokens. For MBPP, a 3-shot prompt template is applied. This baseline is evaluated on HumanEval, MBPP, LiveCodeBench, and ICPC.

Few-shot + CoT. (Wei et al., 2022) This baseline provides the model with few-shot exemplars that demonstrate intermediate reasoning steps in a chain-of-thought style. The prompt includes task-specific exemplars, input-output formats, and target problem instructions. Decoding uses temperature 0.0 to enforce deterministic reasoning. The token budget for generation is set to 16,384 tokens to ensure sufficient space for long-form reasoning chains. No iterative refinement or feedback mechanism is applied; only a single forward generation is executed per task.

Best-of-N. (Kang et al., 2025) This baseline samples $N = 20$ candidates using temperature 0.7. Each candidate is generated with a maximum length of 8192 tokens. If any candidate passes all test cases, execution stops early. This method emphasizes sampling-based diversity.

Reflexion. (Shinn et al., 2023) This baseline performs iterative refinement with at most 10 refinement rounds. Each round conducts 2 internal test executions, then the model reflects and regenerates using temperature 0.7. The maximum output length per generation is 8192 tokens. The total number of LLM calls is approximately 20 (generation + reflection).

Self-Refine. (Madaan et al., 2023) This baseline executes a single initial generation, followed by 3 refinement rounds. Generation uses temperature

Table 8: Generalization of PoT across reasoning domains. Baseline scores are from official reports. All models use non-thinking mode unless otherwise noted. Tasks with native environment feedback (PutnamBench, DeepResearch) show substantially larger gains compared to RM-guided tasks, though RRM-7B still provides effective signals.

Domain	Benchmark	Model	Baseline	+PoT	Feedback Type
STEM + General Knowledge	MMLU-Pro	Qwen3-8B	63.4	74.1	RM (RRM-7B)
Non-formal Math Reasoning	AIME-25	Qwen3-8B	20.9	32.3	RM (RRM-7B)
Formal Math Proof	PutnamBench	DeepSeek-Prover-V2-7B	9.0	30.2	Native Environment
Multimodal Math Reasoning	MathVista	Qwen3-VL-8B	53.9	65.4	RM (RRM-7B)
DeepResearch	TriviaQA	Qwen2.5-7B-Instruct	59.7	75.0	Native Environment

Table 9: Ablation study on AIME-25: Impact of Reward Model choice on PoT performance. The backbone is Qwen3-8B in non-thinking mode. “Self-as-RM” denotes the backbone scoring its own solutions. Stronger reasoning RMs yield gains closer to those seen in environment-grounded tasks.

Reward Model Source	Type	Accuracy (%)	Δ vs Baseline
<i>Baseline (No PoT, Qwen3-8B)</i>	-	20.9	-
Self-as-RM (Qwen3-8B)	Policy Self-Check	28.4	+6.5
RRM-7B	Open Weights	32.3	+11.4
RRM-32B	Open Weights	35.1	+14.2
Gemini-3-Pro	Proprietary API	35.8	+14.9
GPT-5	Proprietary API	36.2	+15.3

0.7, feedback uses temperature 0.3. The maximum output length per generation is 2048 tokens. No code execution is performed; the model internally assesses correctness and may stop early.

ToT (Tree of Thoughts). (Yao et al., 2023) This baseline performs tree-structured search with maximum depth 5, beam width 3, and 3 sampled candidates per node. Sampling uses temperature 0.7, and the maximum output length per node is 2048 tokens. Search proceeds via BFS with scoring and pruning.

LDB (Large Language Model Debugger). (Zhong et al., 2024) This baseline segments a program into basic blocks, executes it on a visible test case to collect runtime variable states, and queries the LLM once per debugging iteration with all selected block-level execution traces; the prompt is capped at 3,097 input tokens, at most 10 basic blocks are included per iteration, greedy decoding (temperature 0) is used, and up to 10 debugging iterations are allowed.

RAP (Reasoning via Planning). (Hao et al., 2023) This baseline uses Monte Carlo Tree Search (MCTS) with 20 iterations, sampling exactly 3 actions per expansion, a maximum reasoning depth of 8, and each LLM call (for action or state generation) capped at 2048 tokens with temperature

0.8.

PG-TD (Planning-Guided Transformer Decoding). (Zhang et al., 2023) This baseline performs tree search over LLM-generated code solutions, where each node represents a partial program; at each leaf, it uses beam search ($beam_size = 1$) to complete the program and evaluates it on public test cases to assign a pass-rate reward; the search uses P-UCB with exploration weight $c = 4$, expands the top-3 most likely next tokens per node, and runs for up to 256 total LLM calls (one per rollout); each LLM generation is capped at 1024 tokens and uses temperature 0.7.

LATS (Language Agent Tree Search). (Zhou et al., 2024) This baseline performs at most 10 search iterations, each expanding 3 nodes. Each node is tested using 2 internal executions. Sampling uses temperature 0.7, with a maximum output length of 4096 tokens. Node selection follows UCB1 with exploration weight 1.414, and the search loop consists of selection, expansion, evaluation, and backtracking.

AB-MCTS (Adaptive Branching Monte Carlo Tree Search). (Inoue et al., 2025) This baseline performs tree search over LLM-generated code solutions, where each node represents a candidate program evaluated by execution on visible test cases;

it uses temperature 0.7, allows up to 128 total LLM calls per problem, and dynamically decides at each node whether to generate a new candidate or refine an existing one based on Bayesian posterior updates of observed scores; the method has no fixed branching factor and caps each LLM generation at 1024 tokens.

RethinkMCTS. (Li et al., 2025) This baseline applies MCTS with 16 rollouts, 3 expansions per rollout, and up to 2 rethink attempts per failed branch. Sampling uses temperature 0.7. The maximum decoding length per expansion is 4096 tokens. A variable P-UCT policy is used with constants 4.0 and 10.0.

CodeT. (Chen et al., 2023a) This baseline generates 10 candidate solutions and synthesizes 5 test suites with 5 test cases each. Sampling uses temperature 0.8, and the maximum output length per candidate is 16384 tokens. Candidate ranking uses Dual Agreement based on test execution outcomes.

Fairness Policy. All baselines are executed under the same hardware, inference backend, timeout configuration, and dataset handling pipeline. Token generation limits, sampling configurations, and evaluation methodologies are fully disclosed to ensure transparent and reproducible comparison.

Our Setting. Our method adopts a structured search paradigm with explicit tree expansions. At most $M = 20$ expansions are allowed, each expansion generates up to $k = 3$ child candidates, and each candidate receives up to 2048 tokens. This yields a maximum search capacity of 60 generated nodes under the same infrastructure as baselines.

Response Length and Budget Fairness. A natural question is whether the different per-candidate token limits across methods (e.g., 8192 for Best-of-N vs. 2048 per node for PoT) constitute an unfair comparison. We argue that fairness should be defined at the level of *total computational budget* rather than per-call token length, for the following reasons.

First, different methods have fundamentally different inference structures. Best-of-N requires each candidate to be a *standalone, complete* solution generated from scratch, so it needs a large per-candidate budget. In contrast, PoT operates as a multi-round search-and-adapt process: each node builds upon previously explored context, and

the MCTS tree structure allows the model to continue refining a solution across iterations rather than regenerating it from scratch each time. This is analogous to how a human programmer debugs iteratively—each attempt does not restart from zero but builds on prior understanding. The per-node token limit reflects this iterative nature: a 2048-token budget per node is sufficient because the model’s adapter has already internalized feedback from earlier attempts.

Second, controlling total budget is strictly more meaningful. Under our protocol, PoT’s total token budget is $20 \times 3 \times 2048 = 122,880$ tokens, while Best-of-N uses $20 \times 8,192 = 163,840$ tokens—a *33% larger budget for Best-of-N*. Enforcing identical per-call limits would either inflate PoT’s total budget (favoring PoT) or compress Best-of-N’s per-candidate budget (favoring PoT even more), neither of which constitutes a fair comparison.

Third, we verify empirically that per-candidate length is not the source of PoT’s advantage. We ran a unified 2048-token cap across all methods on LCB V6:

Table 10: Unified 2048-token cap experiment on LCB V6. Capping all methods to the same per-candidate token limit produces minimal changes and preserves the original ranking.

Method	Original (%)	Unified 2048-cap (%)
Best-of-N	37.71	36.86
AB-MCTS	36.00	36.27
RethinkMCTS	44.12	43.57
PoT (Ours)	49.71	48.94

The ranking is unchanged, and all methods experience comparably small drops (<1 point). This confirms that PoT’s gains arise from its policy evolution mechanism, not from differences in per-candidate token budgets.

Finally, our compute efficiency analysis (Table 18) provides a more comprehensive fairness measure: PoT achieves the best accuracy while consuming only 3.86 PFLOPs per problem, which is *lower* than most search-based and TTT baselines (5–7 PFLOPs). This demonstrates that PoT is not only fair but actually more compute-efficient than the methods it outperforms.

C Robustness Analysis

Multi-seed Stability. Table 11 reports 8-run mean and standard deviation on LCB V6. PoT achieves the highest mean accuracy (49.76%) with the low-

Table 11: Multi-seed evaluation on LCB V6 (8 independent runs with different random seeds).

Method	8-run Mean $\pm$ Std. (%)
Best-of-N	37.64 $\pm$ 0.61
RethinkMCTS	44.09 $\pm$ 0.47
PoT (Ours)	49.76 $\pm$ 0.34

Table 12: Degenerate GRPO group analysis on LCB V6. A group is degenerate when all $k=3$ candidates receive identical rewards, producing zero gradient signal.

Temperature	Degen. Ratio (%)	Acc. (%)
0.2	16.8	56.8
0.4	14.2	57.3
0.6	12.9	57.7
0.8	15.1	57.0

est variance (± 0.34), confirming that the performance gains are robust and not artifacts of seed selection.

Degenerate Group Analysis. Table 12 examines how often all candidates in a GRPO group receive identical rewards (producing no gradient signal). The degenerate ratio remains moderate (12.9–16.8%) across temperatures, and accuracy stays stable. Importantly, even when a degenerate group occurs, subsequent exploration still conditions on accumulated context, so the overall search-and-adaptation process does not stall.

Ablation on LoRA Hyperparameters. Table 5 identifies the stability-performance frontier through joint analysis of rank r and learning rate η . The ($r = 8, \eta = 10^{-4}$) configuration achieves the highest accuracy (49.71%), outperforming others by 2.28–6.28%. Reducing either r or η leads to drops up to 6.28%, suggesting insufficient capacity for immediate knowledge internalization. Increasing r to 16 yields marginal gains but risks overfitting on sparse per-instance feedback. Notably, our approach exhibits a form of *benign overfitting*: despite performing intensive, instance-specific updates using only limited execution signals, the model avoids catastrophic forgetting or policy collapse. Instead, it significantly improves task performance while preserving its base reasoning capabilities. This indicates that PoT’s transient adaptation mechanism effectively channels overfitting toward productive refinement rather than degradation. **Per-Instance Reset vs. Persistent Adapter.** A natural question is whether the transient LoRA adapter should be discarded after each problem or retained across instances. We evaluate a *no-reset* variant

on LCB V6 where the adapter is carried over between consecutive problems. This variant achieves 44.3%, compared with 49.71% for the standard per-instance reset, a drop of 5.41 points. The degradation suggests that instance-specific updates are not reliably transferable across problems and can introduce cross-instance interference. This validates our design choice of per-instance transient adaptation: the adapted policy is a means toward solving the current problem, not a persistent artifact.

D Computational Budget of POT

To ensure reproducibility and transparency regarding the computational footprint of our method, we explicitly define the resource constraints for the inference search process and the on-the-fly training updates used in POT.

D.1 Evaluation Protocol and Fairness Constraints

We adopt pass@1 as the primary evaluation metric across all experiments.

To ensure rigorous comparison, we align PoT with search-based and TTT baselines under a consistent computational budget by enforcing the same node expansion limit.

For code generation benchmarks, the test cases used for reward computation during PoT’s search and adaptation are strictly disjoint from the hidden test cases used for final pass@1 evaluation. The model never accesses hidden evaluation tests during inference, search, or parameter updates.

Additional implementation details and hyperparameter settings are provided below.

D.2 Computational Budget of POT

Inference Budget. For each problem instance, we set a maximum budget of $M = 20$ MCTS iterations. In every iteration, the tree expands by $k = 3$ child nodes, resulting in a theoretical maximum of 60 generated nodes per problem. Each node generation is constrained by a local token limit of $L_{node} = 2048$. Although this search configuration defines a high upper bound, the average path depth in practice is effectively regulated by the objective-driven nature of the MCTS selection policy.

On-the-fly Training Overhead. The core mechanism of POT involves integrating online GRPO updates directly during the search. We utilize LoRA (rank $r = 8$) to update the policy weights efficiently. As demonstrated in our empirical profiling

in Table 13, the computational cost is manageable. For a batch of $k = 3$ nodes (at a representative sequence length), a single LoRA backward pass incurs approximately $1.46\times$ the latency of the corresponding forward pass. The total budget approximation is given by:

$$\begin{aligned} \text{Total Budget} &\approx M \times (\text{Cost}_{\text{fwd},k} + \text{Cost}_{\text{bwd}}) \\ &\approx M \times (1 + 1.46) \times \text{Cost}_{\text{fwd},k=3} \end{aligned} \quad (10)$$

This relationship indicates that the additional overhead for online training is roughly equivalent to adding about 1.5 extra node expansion steps (of width $k = 3$) per iteration.

Efficiency via Fast Convergence: A Case Study.

Although the backward pass increases the per-step cost, POT significantly reduces the total number of iterations required to reach the optimal solution. Consider *Jump Game II*, where the model must overcome a strong prior for $O(N^2)$ dynamic programming in favor of an $O(N)$ greedy approach:

- **Static Baseline Scenario:** Without online adaptation, the search policy often remains trapped in local optima, requiring an extensive budget (e.g., $M_{\text{static}} \approx 104$) to stochastically sample the correct path using only forward expansions.
- **POT Scenario:** By performing gradient updates on failed candidates, POT suppresses sub-optimal reasoning and converges within a much stricter budget, typically requiring far fewer steps (e.g., $M_{\text{POT}} \approx 15$).

The computational efficiency gain is calculated by comparing the total cost of static iterations against the cost-augmented POT iterations:

$$\begin{aligned} \frac{\text{Cost}_{\text{Static}}}{\text{Cost}_{\text{POT}}} &\approx \frac{104 \times \text{Cost}_{\text{fwd},k=3}}{15 \times \text{Cost}_{\text{total},k=3}} \\ &= \frac{104 \times 192.66}{15 \times 473.66} \approx 2.82 \times \end{aligned} \quad (11)$$

This demonstrates that despite the training overhead, POT achieves superior end-to-step efficiency by preventing wasted exploration in suboptimal reasoning branches.

Global Efficiency vs. Performance. As demonstrated in Table 14, POT achieves a state-of-the-art accuracy of 98.78%, significantly outperforming all baselines. Crucially, our reported time (196.42s)

is a total end-to-end duration—strictly encompassing both the LoRA-based backward updates and the MCTS search process—whereas baseline figures (e.g., LATS and AB-MCTS) account for inference only.

Despite including the full training overhead, POT maintains a total latency comparable to, or even lower than, several leading search-oriented approaches like RethinkMCTS (223.27s). This results in a superior efficiency-performance trade-off: by leveraging gradient-driven adaptation to prune search spaces and suppress sub-optimal reasoning early, POT successfully transmutes backward computation into high-precision reasoning. Compared to static search methods that often plateau around 90% accuracy even with extensive budgets, POT demonstrates that on-the-fly learning is a more effective path toward solving complex reasoning tasks without incurring prohibitive total time costs.

E Implementation Details

In this section, we detail the experimental configurations and hyperparameter settings for the PoT framework. Our system consists of a tree-based search mechanism for test-time scaling and a group-relative policy optimization (GRPO) framework for alignment.

E.1 MCTS Search and Code Verification

To enhance the model’s reasoning trajectory exploration, we implement a Monte Carlo Tree Search (MCTS) strategy. The search process is governed by the Upper Confidence Bound (UCB) formula with an exploration constant $c_{\text{puct}} = 1.414$.

For each simulation, the model generates $k = 3$ candidate continuations (generation batch size) with a maximum search depth of 8. The generation process is constrained to a maximum of 2048 new tokens per node with a sampling temperature of $T = 0.7$. To ensure the technical correctness of the synthesized code, we integrate the LiveCodeBench environment for real-time execution-based verification. We set a code execution timeout of 10 seconds and assign a reward weight of 1.0 for successful test case completion. Furthermore, the system incorporates an error feedback mechanism, where execution errors are utilized to refine the search process. The detailed hyperparameters are summarized in Table 15.

Table 13: Profiling of computational cost components for POT ($k = 3$). Total cost reflects the real-time latency as defined in Table 4, where $\text{Cost}_{total} = \text{Cost}_{fwd} + \text{Cost}_{bwd}$.

Component (Batch Size $k = 3$)	Computational Cost (ms)
Forward Pass (Cost_{fwd})	192.66
LoRA Backward (Cost_{bwd} , Rank $r = 8$)	281.00
Total Iteration Cost	473.66
Overhead Ratio ($\text{Cost}_{bwd}/\text{Cost}_{fwd}$)	1.46×

Table 14: Efficiency and performance comparison on HumanEval using Qwen3-4B-Instruct-2507. For PoT, Avg. Time includes both LoRA training and inference; for other baselines, inference only.

Method	Training Req.	Avg. Time (s)	Accuracy (%)
<i>Standard Prompting & Ensemble</i>			
Zero-shot	No	28.51	84.76
Few-shot CoT	No	52.13	85.98
Best-of-N	No	113.87	90.85
<i>Iterative Refinement</i>			
Self-Refine	No	58.96	84.15
Reflexion	No	61.29	87.80
CodeT	No	56.34	88.41
LDB	No	71.23	87.19
<i>Search-based Approaches</i>			
RAP	No	162.33	89.63
PG-TD	No	178.26	89.02
ToT	No	121.46	90.24
LATS	No	178.83	87.80
AB-MCTS	No	189.24	90.85
RethinkMCTS	No	223.27	89.02
PoT (Ours)	Yes (LoRA)	196.42	98.78

E.2 Training and GRPO Parameters

The model is fine-tuned using Group Relative Policy Optimization (GRPO). For each input prompt, we sample a group of $G = 3$ outputs to estimate the advantage function through relative reward ranking, thereby eliminating the need for a separate value network.

The training objective incorporates a KL divergence penalty with a coefficient $\beta = 0.02$ to ensure policy stability. Additionally, a fixed KL coefficient of 0.005 is applied as a baseline constraint. To prevent excessive drift from the initial policy, the reference model is synchronized every 10 update steps. We adopt Parameter-Efficient Fine-Tuning (PEFT) via Low-Rank Adaptation (LoRA) with a rank $r = 8$ and a corresponding vLLM backend configuration. The optimization is conducted using a learning rate of 1×10^{-4} with a PPO clip

Table 15: Hyperparameters for MCTS Search and Code Verification.

Component	Parameter	Value
MCTS Core	Number of simulations	20
	Temperature (T)	0.7
	Maximum search depth	8
	Generation batch size (k)	3
	Max new tokens	2048
Code Evaluation	Code verification	Enabled
	Code reward weight	1.0
	Code timeout (seconds)	10
	Error feedback	Enabled
Selection	Exploration constant (c_{puct})	1.414

ratio $\epsilon = 0.3$ over 3 epochs. The complete training hyperparameters are provided in Table 16.

Table 16: Hyperparameters for GRPO Training and LoRA.

Component	Parameter	Value
Optimization	Learning rate	1×10^{-4}
	PPO epochs	3
	PPO clip ratio (ϵ)	0.3
LoRA (PEFT)	LoRA rank (r)	8
	vLLM max LoRA rank	8
GRPO	Group size (G)	3
	KL penalty coefficient (β)	0.02
	Fixed KL coefficient	0.005
	KL ref update frequency	10

E.3 Computing Infrastructure

To ensure the reproducibility of our results, we detail the hardware and software environment used for both training and inference. All experiments were conducted on a single-node workstation equipped with a single NVIDIA A800 (80GB) GPU. The generous 80GB of HBM2e memory allows for efficient execution of Group Relative Policy Optimization (GRPO) and accommodates the significant KV cache demands during deep MCTS simulations.

The software stack is built upon the latest industry standards to leverage high-performance kernels. We utilize CUDA 13.0 and PyTorch 2.9.0 as the primary deep learning framework. For the test-time scaling phase, we employ vLLM (v0.13.0) as the inference engine, which provides optimized support for the LoRA adapter and high-throughput continuous batching during tree search.

Table 17: Hardware and software specifications.

Component	Specification
GPU	1× NVIDIA A800 (80GB HBM2e)
VRAM	80 GB
Host Memory	128 GB DDR4 (suggested)
Operating System	Ubuntu 22.04 LTS
CUDA Version	13.0
Deep Learning Framework	PyTorch 2.9.0
Inference Engine	vLLM v0.13.0

F Benchmark Details

Our evaluation encompasses five code generation benchmarks with a total of 836 programming problems. The detailed composition of our test suite is as follows:

- **HumanEval:** Contains 164 programming problems designed to evaluate code generation capabilities on diverse algorithmic tasks.
- **MBPP (Mostly Basic Python Problems):** Comprises 257 problems focusing on basic Python programming skills and common coding patterns.
- **LiveCodeBench:** Utilizes the official `code_generation_lite` versions for both v5 and v6 releases:
 - LiveCodeBench v5 lite: 167 problems
 - LiveCodeBench v6 lite: 175 problems
- **ICPC:** A subset of OJBench containing 73 competitive programming problems from International Collegiate Programming Contest scenarios.

In total, our comprehensive benchmark suite contains $164 + 257 + 167 + 175 + 73 = 836$ programming problems across different difficulty levels and problem domains, providing a thorough evaluation of code generation performance across various coding challenges and paradigms.

Table 18: Compute efficiency comparison on LCB V5 using Qwen3-4B-Instruct-2507. All methods use the same evaluation budget (20 iterations, $k=3$, max 2048 tokens). Reported time and FLOPs include full end-to-end cost (forward + backward + search).

Method	Acc. (%)	Avg Time	Est. FLOPs
Best-of-N	43.61	36.4 s	3.02 PF
ToT	43.11	96.8 s	5.94 PF
AB-MCTS	39.52	109.7 s	6.72 PF
RethinkMCTS	49.83	101.3 s	6.08 PF
TTRL	48.61	84.6 s	5.96 PF
TTT-Discover	51.38	78.9 s	5.21 PF
PoT (Ours)	57.49	61.7 s	3.86 PF

Table 19: Results for Qwen3 Thinking variants on LCB V6 benchmark.

Method	LCB V6
Qwen3-4B-Thinking-2507	
+zero-shot	55.20
+PoT (Ours)	63.70
Qwen3-30B-Thinking-2507	
+zero-shot	76.80
+PoT (Ours)	84.90

G Compute Efficiency Analysis

Table 18 compares the compute efficiency of representative methods on LCB V5. Despite including the full backward-pass overhead for LoRA updates, PoT achieves the best accuracy (57.49%) while consuming only 3.86 PFLOPs per problem—less than all search-based and TTT baselines except Best-of-N. This demonstrates that gradient-driven policy evolution effectively prunes the search space, converting backward computation into higher-precision reasoning rather than wasted exploration.

H Additional Results on Thinking Models

To further examine whether PoT complements models with built-in long-form reasoning, we evaluate it on two dedicated thinking models: Qwen3-4B-Thinking-2507 and Qwen3-30B-Thinking-2507.

As shown in Table 19, PoT consistently improves both models. On Qwen3-4B-Thinking-2507, PoT increases LCB V6 accuracy from 55.20 to 63.70 (+8.50 points). On Qwen3-30B-Thinking-2507, accuracy improves from 76.80 to 84.90 (+8.10 points).

These gains indicate that PoT is complementary to native reasoning capabilities rather than a substi-

tute. Even when the base model already performs extended multi-step reasoning, test-time policy evolution remains beneficial by refining reasoning trajectories using environment feedback and adapting the policy to the current instance.