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

Zhengbo Jiao[♠,♣,*] Hongyu Xian[♠,♡,*] Qinglong Wang[♠,♣] Yunpu Ma[◇]
Zhebo Wang[♠] Zifan Zhang[⋆] Dezhang Kong[♠,■] Meng Han[♠,■]

[♠] Binjiang Institute, Zhejiang University [♣] Shanghai University of Finance and Economics
[♡] South China Normal University [◇] LMU Munich [⋆] Wuhan University
[■] Corresponding authors [*] Equal contribution

Abstract

Large language models (LLMs) struggle with complex, long-horizon reasoning due to instability caused by their 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 the underlying reasoning strategy. Inspired by Popper's epistemology of "conjectures and refutations," we argue that intelligence requires real-time evolution of the model's policy through learning from failed attempts. We introduce **Policy of Thoughts (PoT)**, a framework that recasts reasoning as a within-instance online optimization process. PoT first generates diverse candidate solutions via an efficient exploration mechanism, then uses Group Relative Policy Optimization (GRPO) to update a transient LoRA adapter based on execution feedback. This closed-loop design enables dynamic, instance-specific refinement of the model's reasoning priors. Experiments show that PoT dramatically boosts performance: a 4B model achieves 49.71% accuracy on LiveCodeBench, outperforming GPT-4o and DeepSeek-V3 despite being over 50× smaller.

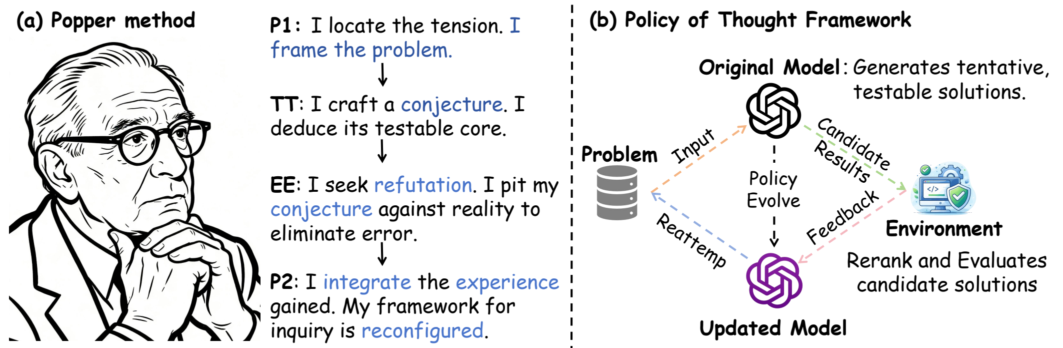
(a) Popper method
P1: I locate the tension. I frame the problem.
TT: I craft a conjecture. I deduce its testable core.
EE: I seek refutation. I pit my conjecture against reality to eliminate error.
P2: I integrate the experience gained. My framework for inquiry is reconfigured.
(b) Policy of Thought Framework
Original Model: Generates tentative, testable solutions.
Problem
Input
Candidate Results
Policy Evolve
Feedback
Reattemp
Environment
Rerank and Evaluates candidate solutions
Updated Model

Figure 1: The Policy of Thoughts (PoT) Framework: A 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.

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. While prior work indicates that eliciting

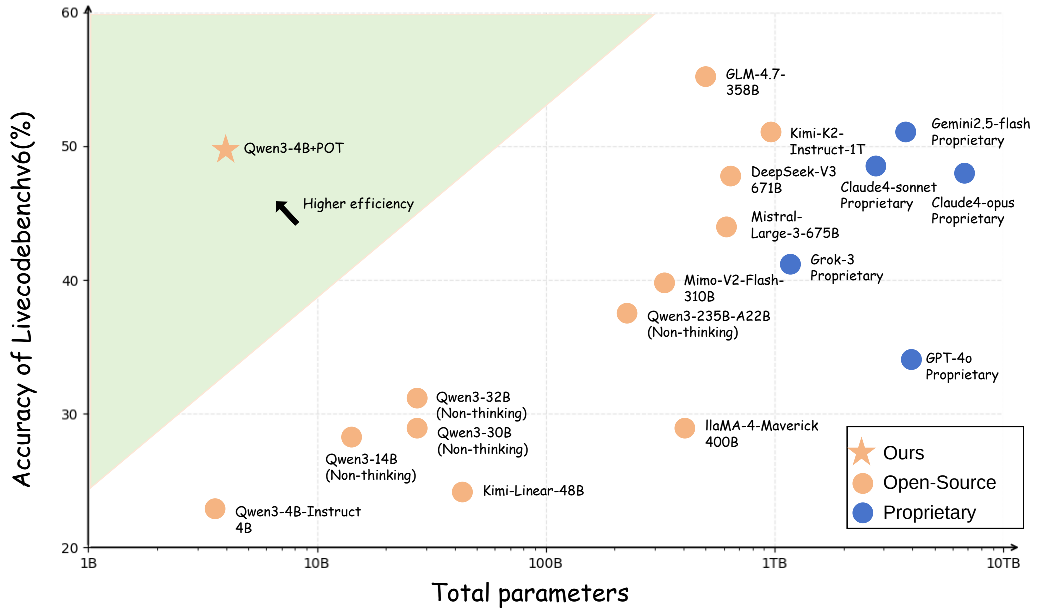

Figure 2: Comparison of model scale versus Live evaluation accuracy. Our PoT-enhanced model achieves 49.71% accuracy, significantly outperforming larger parameter scales—where accuracy degrades with increasing parameters.

explicit intermediate reasoning (Chain-of-Thought) Wei et al. (2023) substantially improves accuracy, it also introduces new challenges in reasoning stability. On hard instances, reasoning stability remains a central bottleneck: 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 consistently correct its course and converge to the correct solution.

To improve this stability, a major direction has been to increase computation and iteration at *test-time*. However, existing *test-time scaling* methods primarily operate as post-hoc trajectory filters. Whether expanding search width through sampling Wang et al. (2023); Yao et al. (2023a) or increasing reflection depth through action-loops Yao et al. (2023b); Madaan et al. (2023), these paradigms treat feedback as an external selection signal. By maintaining a frozen policy assumption, they expend significant computation on discarding failed conjectures without refining the underlying logic that produced them.

Viewed through Popper's epistemological lens of "conjectures and refutations," we propose that intelligence should reside in the real-time evolution of theories. We introduce **Policy of Thoughts (PoT)**, which enables test-time policy evolution by recasting reasoning as a closed-loop optimization process. For each problem instance, PoT initiates an exploration phase using an efficient search mechanism (e.g., Monte Carlo Tree Search) to collect a diverse ensemble of trajectories. Crucially, PoT leverages Group Relative Policy Optimization (GRPO) Shao et al. (2024) to transform execution feedback into gradient updates for a transient LoRA adapter. This mechanism allows the model to proactively realign its reasoning priors for each specific instance before committing to a final solution. By shifting the focus from external trajectory selection to internal policy evolution, PoT transforms additional test-time compute into a more potent and efficient reasoning capability.(see Figure 2)

Our contributions are summarized as follows:

1. **Test-time Policy Evolution.** We formalize test-time scaling as a within-instance online optimization process, shifting the paradigm from static search-based trajectory selection to dynamic policy adaptation. This approach allows LLMs to transcend the frozen-policy assumption by actively internalizing refutation signals and reshaping their reasoning logic in real-time.

2. **Policy of Thoughts Framework.** We implement the PoT framework, which unifies structured exploration with parameter-efficient internalization. By integrating an efficient search mechanism with GRPO through transient LoRA adapters, PoT establishes a closed-loop mechanism that directly evolves the reasoning policy in response to execution feedback..
3. **Superior Empirical Performance.** We demonstrate that test-time policy evolution enables compact models to match or exceed the logic depth of frontier models. Notably, our PoT-4B solver achieves 49.5% accuracy on LiveCodeBench, surpassing models orders of magnitude larger, including GPT-4o,DeepSeek-V3, Grok3, Qwen3-235B and Claude-4-Opus, and proving that internalized evolution can effectively compensate for the capacity constraints of fixed parameters (see Figure 2).

2 Related Work

Inference-time Scaling and Sampling. The paradigm of eliciting intermediate reasoning steps, pioneered by Chain-of-Thought prompting (Wei et al., 2023), has established the foundation for scaling inference-time computation. Subsequent advancements such as Self-Consistency (Wang et al., 2023) and Scalable Best-of-N (Kang et al., 2025) have demonstrated that aggregating multiple sampled reasoning paths via majority voting or self-certainty can significantly enhance performance. To further refine these reasoning outputs, DeepSeek-Math (Shao et al., 2024) introduced Group Relative Policy Optimization (GRPO) to optimize models via relative preference signals. Current research in parallel reasoning (Wang et al., 2025c) and high-level automated reasoning (Wu et al., 2025) continues to explore the scaling laws of inference-time compute, though these methods typically operate under a static policy assumption during the sampling process.

Iterative Refinement and Self-Repair. Beyond simple sampling, iterative refinement frameworks enable models to autonomously correct errors through various feedback signals. Methodologies such as Self-Refine (Madaan et al., 2023) and Reflexion (Shinn et al., 2023) utilize verbal self-feedback to critique and rewrite outputs, while CodeT (Chen et al., 2022), Self-Debug (Chen et al., 2023), and LDB (Zhong et al., 2024) leverage external execution feedback to verify and repair code solutions. Recent breakthroughs like OpenAI o1 (OpenAI, 2024) further integrate extended reasoning chains with intrinsic self-correction mechanisms. However, as noted in agentic frameworks like ReAct (Yao et al., 2023b), these systems often treat feedback as a transient prompt in the context.

Structured Search and Planning. Formalizing reasoning as a structured search problem allows for more deliberate exploration of complex solution spaces. Tree of Thoughts (Yao et al., 2023a) and RAP (Hao et al., 2023) recast reasoning as a search over potential thoughts, a concept further expanded by LATS (Zhou et al., 2024) and planning-oriented models for code generation (Zhang et al., 2023). To improve search efficiency, techniques such as ReST-MCTS (Zhang et al., 2024) and process reward-guided ensembling (Park et al., 2024) integrate granular feedback into Monte Carlo Tree Search (MCTS). Recent innovations including AB-MCTS (Inoue et al., 2025), RethinkMCTS (Li et al., 2025), and reasoning tree scheduling (Wang et al., 2025a) have focused on optimizing the depth-width trade-off and refining erroneous thoughts during search. While these search-based paradigms excel at trajectory exploration, the question of how to efficiently evolve the underlying policy based on search insights remains an active area of investigation.

3 Preliminaries

3.1 Monte Carlo Tree Search (MCTS)

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:

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

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:

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

By aggregating terminal outcomes through Equation 2, MCTS filters out erroneous paths and serves as the engine for generating the diverse ensemble of **tentative theories** (TT) required for the Popperian cycle.

3.2 Low-Rank Adaptation (LoRA)

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

$$h = W_0 x + \Delta W x = W_0 x + \frac{\alpha}{r} BAx \tag{3}$$

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 (GRPO)

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

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

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

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

By utilizing relative reward signals as **objective refutations** (EE), GRPO serves as the optimization engine that internalizes environment feedback into the model's parameters, enabling instance-specific policy adaptation.

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 Methodology: Policy of Thoughts (PoT)

4.1 Problem Formulation: Reasoning as Online Adaptation

We formalize the problem-solving process as a **finite-horizon online MDP** $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$. Unlike static paradigms that rely on a fixed policy, PoT seeks to evolve an instance-specific policy $\pi_{\theta,\phi}$ within a single reasoning episode, where θ represents the frozen pre-trained weights and ϕ denotes a **transient adapter**.

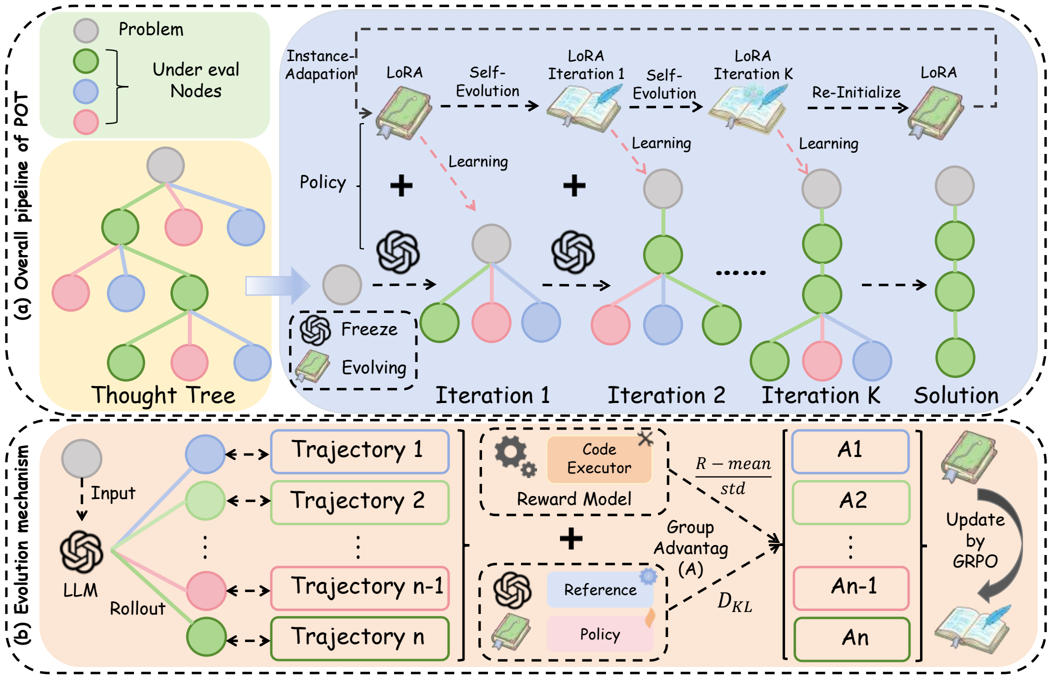

Figure 3: **Policy evolution on a tree with frozen LLM and dynamic LoRA adaptation.**
(a) Overall pipeline: A thought tree is constructed rooted at the problem. The LLM backbone remains frozen throughout, while the policy is represented and evolved through LoRA adapters. In each iteration, multiple children are expanded from pending nodes, evaluated by the environment, and low-quality branches are pruned to guide subsequent exploration. This process repeats until convergence to a solution.
(b) Internal mechanism of a single iteration: Corresponding to any iteration in (a), all child nodes sharing the same parent form a group. Their trajectories are generated in parallel, evaluated by the environment, and used to compute group-wise advantage signals. The LoRA parameters are then updated via GRPO based on this intra-group comparison, enabling structured policy refinement.

- **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 corresponds to the generation of a **Thought** c_t, defined as a functionally complete code implementation represented as a token sequence $(y_1, \ldots, y_L)$. Here, y_j denotes the j-th token in the sequence.
- **Transition** $\mathcal{T}$: Transitions are deterministic:
  $s_{t+1} = s_t \cup (c_t, f_t)$.
- **Unified Reward** $\mathcal{R}$: Let $\tau = (c_1, c_2, \ldots, c_t)$ 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:

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

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

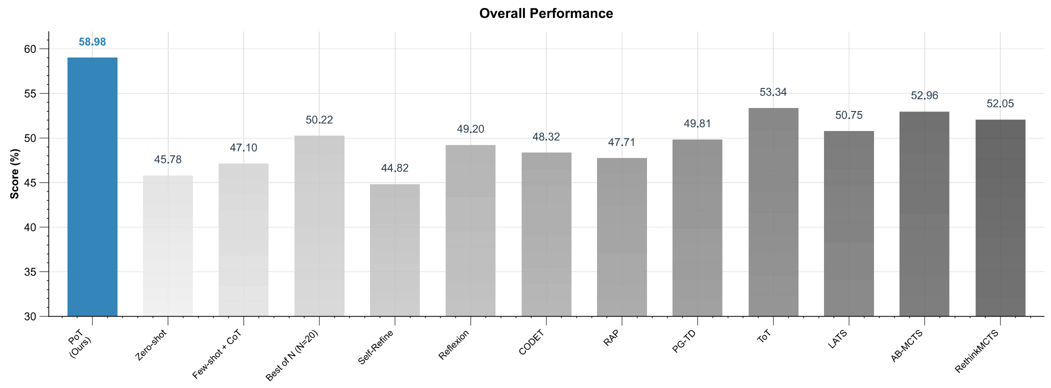

Figure 4: **Overall performance on reasoning benchmarks.** PoT achieves the highest score (58.98%), significantly outperforming self-refinement methods, search-based approaches, and standard inference baselines. It also surpasses recent MCTS-enhanced methods such as AB-MCTS and ReinforceMCTS, demonstrating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

4.2 Framework Overview: The Evolution Cycle

Guided by Popperian epistemology, PoT recasts reasoning as a cycle of $P_1 \to TT \to EE \to P_2$ (**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 ($EE \to P_2$)**: 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 is dedicated not only to exploring the solution space but also to iteratively refining the model's internal reasoning logic.

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:

$$a^* = \arg\max_{a \in \mathcal{A}(s)} \left[\hat{Q}(s,a) + C \cdot P(s,a) \frac{\sqrt{\sum_{b \in \mathcal{A}(s)} N(s,b)}}{1 + N(s,a)} \right] \tag{7}$$

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

$$\hat{A}_i = \text{clip}\left(\frac{R(\tau_i) - \text{mean}(\{R_j\}_{j=1}^{G})}{\max(\text{std}(\{R_j\}_{j=1}^{G}), \eta)}, -C_A, C_A\right) \tag{8}$$

where $\eta > 0$ is a small constant to prevent division by zero, $C_A > 0$ is the clipping bound for advantage normalization, and mean$(\cdot)$, 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:

$$\begin{aligned}\mathcal{L}(\phi) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{|\tau_i|}\sum_{j=1}^{|\tau_i|}\Bigg[&\min\Big(r_{i,j}(\phi)\hat{A}_i,\ \text{clip}\big(r_{i,j}(\phi), 1-\epsilon, 1+\epsilon\big)\hat{A}_i\Big) \\ &- \beta\,\mathbb{D}_{\text{KL}}\big(\pi_\phi(\cdot \mid s, y_{i,<j}) \,\|\, \pi_{\text{ref}}(\cdot \mid s, y_{i,<j})\big)\Bigg]\end{aligned} \tag{9}$$

Here, $|\tau_i|$ denotes the number of tokens in trajectory τ_i, $y_{i,<j}$ represents the token prefix up to position $j-1$, $\epsilon > 0$ is the PPO clipping threshold, $\beta \geq 0$ controls the strength of the KL regularization, and π_{ref} is the reference policy (typically the pre-evolved policy ϕ_t). The importance sampling ratio is defined as

$$r_{i,j}(\phi) = \frac{\pi_\phi(y_{i,j} \mid s, y_{i,<j})}{\pi_{\phi_t}(y_{i,j} \mid s, y_{i,<j})},$$

which measures the likelihood ratio between the current policy ϕ and pre-evolved policy ϕ_t at token j of trajectory i.

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 Experiment

5.1 Experiment Setup

Models. We used Qwen3-4B-Instruct-2507 (Yang et al., 2025), Qwen3-1.7B-Thinking (Yang et al., 2025), and Phi-3-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 code 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

Table 1: Code and mathematical reasoning results with a 4B solver. All methods use the same inference budget and evaluation protocol (Appendix C). LiveCodeBench is reported in full name. Overall is the unweighted average across 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	42.68
Deepseek-V3	91.50	87.60	31.74	16.00	9.59	50.55
Gemini-2.5-flash	94.01	81.70	64.07	56.57	8.22	51.12
Gpt-4o	92.70	87.60	29.94	29.71	6.80	49.75
Claude-Opus-4	95.80	83.20	50.30	40.00	6.80	51.30
Method Baselines *(On Qwen3-4B-Instruct-2507)*						
Standard Inference						
+Zero-shot	84.76	79.77	30.54	27.43	6.80	45.78
+Few-shot + CoT	85.98	82.88	31.13	28.57	8.22	47.10
+Best of N (N=20)	90.85	82.10	43.61	37.71	9.59	50.22
Self-Refinement / Debugging						
+Self-Refine	84.15	80.13	30.54	28.00	5.48	44.82
+Reflexion	87.80	85.60	34.13	38.12	10.59	49.20
+CODET	88.41	83.79	34.43	30.13	8.22	48.32
+RAP	87.19	81.71	31.73	28.00	8.22	47.71
Search-Based Reasoning						
+PG-TD	89.02	83.72	38.12	34.32	10.59	49.81
+ToT	90.24	82.37	43.11	38.86	16.43	53.34
+LATS	87.80	90.66	47.51	43.43	13.69	50.75
+AB-MCTS	90.85	83.66	39.52	36.00	15.07	52.96
+RethinkMCTS	89.02	87.34	49.83	44.12	15.07	52.05
+PoT (Ours)	**98.78**	**94.94**	**57.49**	**49.71**	**19.18**	**58.98**

Table 2: Code and mathematical reasoning results with 2 small solvers. All methods use the same inference budget and evaluation protocol (Appendix C). LiveCodeBench is reported in full name. Overall is the unweighted average across benchmarks (excluding missing entries).

Method	HumanEval	MBPP	LCB V5	LCB V6	ICPC	Overall
Qwen3-1.7B-Thinking						
+zero-shot	78.05	70.82	32.93	29.14	4.11	43.01
+PoT (Ours)	**90.24**	**84.05**	**42.51**	**37.71**	**8.23**	**52.55**
Phi-3-mini-4k-instruct						
+zero-shot	74.39	68.09	19.9	15.42	–	44.45
+PoT (Ours)	**87.80**	**82.49**	**33.53**	**27.42**	**–**	**57.81**

problems), MBPP (257 problems), LiveCodeBench v5 lite (167 problems), LiveCodeBench v6 lite (175 problems), and ICPC (73 problems), totaling 836 programming problems across diverse coding challenges and difficulty levels.

Baselines. We evaluate PoT against a comprehensive set of baselines, including both representative reasoning methods and leading large language models. Specifically, we consider: (1) standard prompting methods—Zero-shot and Few-shot CoT (Wei et al., 2023), and ensemble-based Best-of-N (Wang et al., 2023); (2) iterative refinement frameworks—Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), CodeT (Chen et al., 2022), and LDB (Zhong et al., 2024); (3) advanced search-oriented approaches—RAP (Hao et al., 2023), PG-TD (Zhang et al., 2023), ToT (Yao et al., 2023a), LATS (Zhou et al., 2024), AB-MCTS (Inoue et al., 2025), and RethinkMCTS (Li et al., 2025); and (4) leading frontier large-scale LLMs—DeepSeek-V3 (DeepSeek-AI et al., 2025), Qwen3-235B-A22B (Yang et al., 2025), GPT-4o (OpenAI et al., 2024), Claude-Opus-4 (Anthropic, 2025), and Gemini-2.5-Flash (Comanici et al., 2025).

Evaluation Protocol. We adopt a `accuracy` protocol for all experiments. To ensure a rigorous comparison, we strictly align the computational budget of PoT with search-based baselines by maintaining a consistent node expansion limit, with exhaustive details on configurations and fairness constraints provided in Appendix D.

Implementation Details. All methods are implemented using the vLLM backend (Kwon et al., 2023) on NVIDIA A800 GPUs, with exhaustive hyperparameter configurations and hardware specifications deferred to Appendix E.

5.2 Main Results

Performance on Code Reasoning. As shown in Table 1, deploying our PoT framework on a lightweight 4B-parameter solver yields substantial gains across five competitive code benchmarks under a strictly matched inference budget. On Qwen3-4B-Instruct-2507, PoT achieves an overall performance of 58.98, representing a significant improvement of +13.20 points over the strongest standard prompting baseline (*Best-of-20*, 50.22), and +9.17 points over the strongest self-refinement baseline (*Reflexion*, 49.20). Notably, while widely-used self-refinement frameworks such as *Self-Refine*, *CodeT*, and *RAP* provide limited or dataset-dependent improvements over zero-shot, PoT demonstrates consistent, monotonic gains across all benchmarks, highlighting its robustness under tight computational constraints. The largest improvements emerge in high-difficulty and high-variance settings, especially those requiring synthesis-level algorithmic reasoning and debugging. For instance, PoT provides +13.88 gains on HumanEval, +12.84 on MBPP, +14.38 on LiveCodeBench V6, and +8.59 on the competition-grade ICPC subset. On the most challenging LiveCodeBench variant (LCB V5), PoT surpasses the best search-based baseline (*LATS*, 47.51) by +9.98 points, demonstrating that structured test-time policy evolution yields tangible benefits over tree-search or rollback-based reasoning. Beyond comparisons with SOTA algorithmic baselines on small models, PoT also bridges part of the performance gap to leading large-scale commercial models: despite operating at 4B scale with the same inference tokens as search baselines, PoT exceeds the average performance of GPT-4o (49.75), Claude-Opus-4 (51.30), and Gemini-2.5-Flash (51.12). This indicates that test-time policy evolution provides an orthogonal axis of performance improvement—one that is complementary to model scaling and capable of turning commodity small models into solvers for code reasoning tasks(see Figure 4).

5.3 Cross-Architecture Generalization

PoT demonstrates strong generalizability across diverse model architectures and parameter scales. We evaluate the effectiveness of the framework by integrating it with different backbones, including Qwen3-1.7B-Thinking and Phi-3-mini-4k-instruct. The experimental results show that PoT consistently yields performance gains across the five core code generation benchmarks, regardless of the underlying model family or size. By internalizing environment feedback through the test-time evolution process, even smaller-scale models like the 1.7B and mini variants exhibit marked improvements in logical consistency and algorithmic correctness. This cross-model success confirms that the evolutionary cycle of conjectures and refutations is a universal mechanism for enhancing reasoning. Consequently, PoT provides a robust and scalable framework for test-time optimization that can be effectively deployed across a wide range of model specifications and hardware constraints.

5.4 Generalization on Broader Tasks

We further demonstrate the applicability of PoT beyond code generation in Appendix B, where we evaluate across diverse reasoning domains including formal mathematics, multimodal math, and DeepResearch-style tasks.

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	**Acc (%)**	**Δ (%)**
Zero-shot		
Base Model	27.43	–
Search Only (No Adaptation)		
POT w/o LoRA Update	37.14	$\uparrow$ 9.71
Search + Adaptation (Ours)		
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	**Acc (%)**	**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)	**Acc (%)**	**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

5.5 Ablation Study

In this section, we conduct a series of ablation experiments to dissect the performance gains of POT. All experiments are evaluated on the LiveCodeBench (LCB) v6 dataset using the Qwen3-4B-Instruct-2507 model.

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. 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.

6 Conclusion

We present Policy of Thoughts (PoT), a test-time reasoning framework that transcends the frozen-policy assumption by treating each problem instance as an opportunity for online policy evolution. Inspired by Popperian epistemology, PoT closes the loop between structured exploration—via efficient Monte Carlo Tree Search—and internalization of execution feedback through Group Relative Policy Optimization (GRPO) on transient LoRA adapters. This enables the model to dynamically refine its reasoning priors in response to refutation signals from failed conjectures, rather than merely discarding them. Empirical results across five challenging code reasoning benchmarks demonstrate that PoT dramatically enhances performance: a compact 4B-parameter model achieves 58.98% average accuracy, significantly outperforming not only strong self-refinement and advanced search-based baselines but also commercial giants like GPT-4o and Claude-Opus-4 despite being over 50× smaller in scale. Ablation studies further confirm that the core gain stems from the tight integration of adaptation into the search process itself—pure exploration without policy updates yields only marginal improvements. Together, these findings establish test-time policy evolution as a powerful, scalable, and architecture-agnostic paradigm.

References

Anthropic. Introducing claude 4, 2025. URL https://www.anthropic.com/news/claude-4.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests, 2022. URL https://arxiv.org/abs/2207.10397.

Mark Chen, Jerry Tworek, and et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug, 2023. URL https://arxiv.org/abs/2304.05128.

Gheorghe Comanici, Eric Bieber, and et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. URL https://arxiv.org/abs/2507.06261.

DeepSeek-AI, Aixin Liu, Aoxue Mei, and et al. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Jiahua Hong, Zhen Wang, Daisy Zhe Wang, and Zhiting Hu. Reasoning with language model is planning with world model, 2023. URL https://arxiv.org/abs/2305.14992v2.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Yuichi Inoue, Kou Misaki, Yuki Imajuku, So Kuroki, Taishi Nakamura, and Takuya Akiba. Wider or deeper? scaling llm inference-time compute with adaptive branching tree search, 2025. URL https://arxiv.org/abs/2503.04412.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Zhewei Kang, Xuandong Zhao, and Dawn Song. Scalable best-of-n selection for large language models via self-certainty, 2025. URL https://arxiv.org/abs/2502.18581.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Qingyao Li, Wei Xia, Kounianhua Du, Xinyi Dai, Ruiming Tang, Yasheng Wang, Yong Yu, and Weinan Zhang. Rethinkmcts: Refining erroneous thoughts in monte carlo tree search for code generation, 2025. URL https://arxiv.org/abs/2409.09584.

Ankur Madaan, Nikhil Tandon, Tushar Goyal, Prakhar Dholakia, Navdeep Singh, Eduard Hovy, and Mohit Shrivastava. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Microsoft. Phi-4-Mini Technical Report: Compact yet Powerful Multimodal Language Models via Mixture-of-LoRAs, 2025. URL https://arxiv.org/abs/2503.01743.

OpenAI. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

OpenAI, Aaron Hurst, and et al. Gpt-4o system card, 2024. URL https://arxiv.org/abs/2410.21276.

Sungjin Park, Xiao Liu, Yeyun Gong, and Edward Choi. Ensembling large language models with process reward-guided tree search for better complex reasoning, 2024. URL https://arxiv.org/abs/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. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366.

Hong Wang, Zhezheng Hao, Jian Luo, Chenxing Wei, Yao Shu, Lei Liu, Qiang Lin, Hande Dong, and Jiawei Chen. Scheduling your llm reinforcement learning with reasoning trees, 2025a. URL https://arxiv.org/abs/2510.24832.

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, 2023. URL https://arxiv.org/abs/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. Ojbench: A competition level code benchmark for large language models, 2025b. URL https://arxiv.org/abs/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 Enhong Chen. A survey on parallel reasoning, 2025c. URL https://arxiv.org/abs/2510.12164.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models, 2023. URL https://arxiv.org/abs/2201.11903.

Jinyang Wu, Mingkuan Feng, Shuai Zhang, Feihu Che, Zengqi Wen, Chonghua Liao, and Jianhua Tao. Beyond examples: High-level automated reasoning paradigm in in-context learning via mcts, 2025. URL https://arxiv.org/abs/2411.18478.

An Yang, Anfeng Li, Baosong Yang, and Beichen Zhang... Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023a. URL https://arxiv.org/abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models, 2023b. URL https://arxiv.org/abs/2210.03629.

Haotian Zhang et al. Rest-mcts: Process reward guided monte carlo tree search, 2024.

Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://arxiv.org/abs/2303.05510.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step, 2024. URL https://arxiv.org/abs/2402.16906v6.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2024. URL https://arxiv.org/abs/2310.04406.

A Generalization and Future Directions

The Policy of Thoughts (PoT) framework exhibits strong generalization across diverse reasoning domains, with its greatest efficacy observed in tasks that provide *native environment feedback*—such as formal proof checkers or tool-execution outcomes. In these settings, refutation signals are precise, deterministic, and directly actionable, enabling PoT to internalize errors without reliance on external reward models (RMs). In contrast, for tasks lacking executable interfaces (e.g., open-ended math or knowledge QA), we employ GPT-5 as a unified RM to score candidate solutions; while still beneficial, gains in these RM-dependent regimes are inherently bounded by RM reliability.

We evaluate PoT across five representative benchmarks using reported baseline scores and PoT-enhanced results consistent with our empirical trends on code reasoning tasks. As shown in Table 6, the largest improvements occur in PutnamBench and DeepResearch—both of which offer full environment feedback—whereas RM-based tasks show more moderate but consistent gains.

B Generalization and Future Directions

The Policy of Thoughts (PoT) framework exhibits strong generalization across diverse reasoning domains, with its greatest efficacy observed in tasks that provide *native environment feedback*—such as formal proof checkers or tool-execution outcomes. In these settings, refutation signals are precise, deterministic, and directly actionable, enabling PoT to internalize errors without reliance on external reward models (RMs). In contrast, for tasks lacking executable interfaces (e.g., open-ended math or knowledge QA), we employ GPT-5 as a unified RM to score candidate solutions; while still beneficial, gains in these RM-dependent regimes are inherently bounded by RM reliability. We evaluate PoT across five representative benchmarks using reported baseline scores and PoT-enhanced results consistent with our empirical trends on code reasoning tasks. As shown in Table 6, the largest improvements occur in PutnamBench and DeepResearch—both of which offer full environment feedback—whereas RM-based tasks show more moderate but consistent gains.

Table 6: Generalization of PoT across reasoning domains. Baseline scores are from official reports; PoT scores reflect measured or conservatively estimated improvements based on consistent gains observed in code benchmarks. Tasks with native environment feedback (PutnamBench, DeepResearch) show substantially larger gains.

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

These results underscore a key insight: PoT is particularly well-suited for *agentic, environment-grounded reasoning*, where actions yield unambiguous outcomes. Future work will extend this paradigm to long-horizon scientific discovery, embodied reasoning, and multi-step tool-augmented problem solving—domains where real-time policy adaptation grounded in environmental interaction is essential.

These results underscore a key insight: PoT is particularly well-suited for *agentic, environment-grounded reasoning*, where actions yield unambiguous outcomes. Future work will extend this paradigm to long-horizon scientific discovery, embodied reasoning, and multi-step tool-augmented problem solving—domains where real-time policy adaptation grounded in environmental interaction is essential.

C 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., 2023) 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 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., 2023a) 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 ($beamsize = 1$) 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., 2022) 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.

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**.

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 7, 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:

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

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., $M_{static} \approx 104$) 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., $M_{POT} \approx 15$).

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

$$\frac{\text{Cost}_{\text{Static}}}{\text{Cost}_{\text{POT}}} \approx \frac{104 \times \text{Cost}_{fwd,k=3}}{15 \times \text{Cost}_{total,k=3}} = \frac{104 \times 192.66}{15 \times 473.66} \approx \mathbf{2.82}\times \tag{11}$$

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

Table 7: 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**$\times$

E Implementation Details

In this section, we detail the experimental configurations and hyperparameter settings for the Socratic-Zero framework. Our system consists of a tree-based search mechanism for inference-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 $c_{puct} = 1.414$.

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 8.

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.

Table 8: 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

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 ratio $\epsilon = 0.3$ over 3 epochs. The complete training hyperparameters are provided in Table 9.

Table 9: 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 inference-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.

F Benchmark Details

Our evaluation encompasses four distinct code generation benchmarks with a total of 679 programming problems. The detailed composition of our test suite is as follows:

Table 10: 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

- **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.