ZetaEvolve: History-Conditioned Potential Value for Self-Evolving Search

Jiyang Shen [1, 2] **Xiaojing Zhang** [1] **Bochen Lyu** [3, 1,*] **Zhanxing Zhu** [3,*]
[1]DataCanvas [2]Cambricon [3]University of Southampton
jiyangc38@gmail.com, zhangxjohn@outlook.com, {bochen.lyu, z.zhu}@soton.ac.uk

Abstract

Self-evolving agents have shown strong empirical performance in mathematical discovery and code optimization, where the search space is often too large and unstructured for exhaustive exploration. However, many existing methods are designed without an explicit model of the self-evolution process, making it unclear which components and search signals are necessary for effective search. In this paper, we take a node-centered view of self-evolution as a partially observable Markov decision process (POMDP) and use this view to motivate a history-conditioned search heuristic. We treat the context surrounding each search node, including the current program and its evolution-related information, as observable history, while the node's underlying search state remains latent and dynamically shaped by the ongoing search process. This perspective connects parent-node selection, prompt transformation, and child-code generation within a unified sequential decision framework, aligning search behavior more closely with the objective of self-evolution. We then develop **ZetaEvolve**, a search strategy that improves Predictor Upper Confidence bound applied to Trees (PUCT) with a history-conditioned mechanism tailored to partial-observation signals in self-evolving search. Empirically, ZetaEvolve converges faster and reaches stronger solutions than the compared methods.

1 Introduction

Self-evolving agents have shown strong potential in scientific discovery, mathematical reasoning, and optimization [26, 7]. Current paradigms typically formulate the target problem as a program optimization task, in which candidate programs are iteratively generated, executed, evaluated, and refined through guided search.

From the perspective of model and sampling scale, existing self-evolving methods are mainly built on powerful frontier-model APIs, such as AlphaEvolve and its variants [18, 16, 4, 2, 12, 24]. Leveraging strong contextual understanding and reasoning capabilities of Large Language Models (LLM), these methods integrate evolutionary heuristics into the loop and achieve clear improvements over baselines. However, due to high API cost and slow generation speed, they do not easily support large-scale parallel sampling. In contrast, another line of work adopts smaller, locally deployed models [27, 29, 25], enabling faster generation, large-scale parallel sampling, and even online updates during search. However, to maintain efficiency under parallel execution, some approaches reduce context length and rely primarily on single-node information [27]. Moreover, smaller models generally exhibit weaker contextual processing capabilities [3, 15, 6]. As a result, few small-model-based methods effectively incorporate rich historical context into self-evolution.

From the perspective of search strategy, various sampling algorithms have also been explored for self-evolution, with the goal of allocating limited search budget more effectively. Representative

*Corresponding author

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

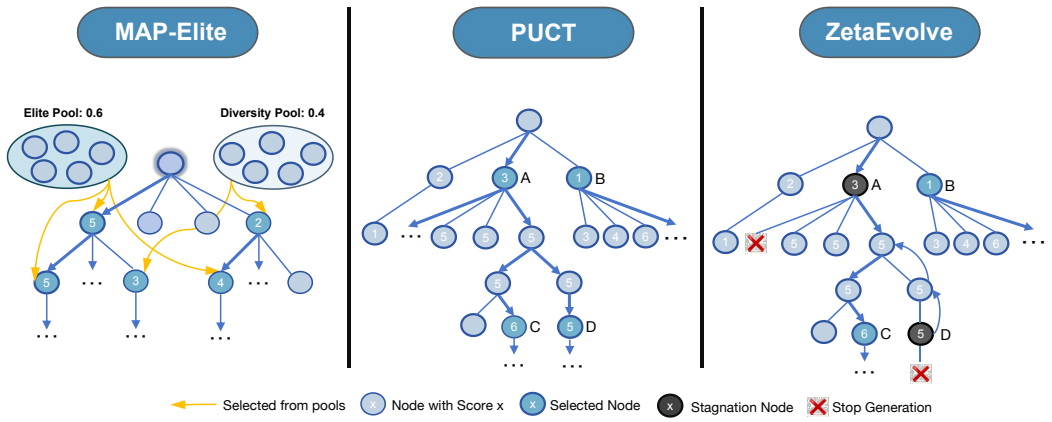


Figure 1: Comparison of search strategies. **Left:** In MAP-Elite, nodes in the evolving tree are assigned into two pools based on evaluation scores. At each step, nodes are selected from pools according to predefined proportions or sampling probabilities (like 0.6, 0.4), and expanded independently. **Middle:** PUCT treats high-child-score nodes (A, B) as exploitation targets and high-score nodes as exploration candidates (C, D). However, if a node (A) continues to generate high-scoring children without improving its best child, or if a trajectory (C) continues to yield strong local scores without surpassing its trajectory-level best, PUCT may keep allocating search budget to branches that are locally unproductive. **Right:** ZetaEvolve selects nodes by history-conditioned potential value (in Eq.(6)). Nodes (A, D) are downweighted when their histories indicate stagnation, so expansion is stopped along these directions (red crosses). Budget is allocated toward nodes such as B and C, where child improvements remain possible, allowing the search to avoid high-scoring but saturated regions.

approaches based on MAP-Elite, such as those in [18, 16, 27], maintain diversity through elite archives and grid maps, but may suffer from slow convergence and unstable results due to high stochasticity. Yuksekgonul et al. [29] adopt Predictor Upper Confidence bound applied to Trees (PUCT) [20, 21] for node selection to balance exploration and exploitation, but since PUCT is not inherently designed for self-evolution settings, its search and sampling behavior is often misaligned with the self-evolution process, leading to limited performance (as analyzed in Sec. 4.1).

Taken together, these limitations reveal a challenge for self-evolution: how to effectively utilize historical context without sacrificing parallel sampling, while making search better aligned with the objective of self-evolution. To address this, we view the local evolution dynamics around each node as a Partially Observable Markov Decision Process (POMDP), where the agent cannot access the full underlying node state and observes only local history, such as the current program, its context programs, the corresponding evaluation scores, and other local information around the node. Under this POMDP view, history becomes a natural decision unit, and the objective of self-evolution is more clearly defined, since the long-term utility of expanding a node depends not only on its current score, but also on the local trajectory and search context around it.

Our framework therefore uses local history as a proxy for the latent state. Rather than directly packing extensive historical context into prompts, we propose **ZetaEvolve**[2], which maps observable node histories into search statistics and uses them to estimate a history-conditioned potential value motivated by the self-evolution objective under the POMDP view. As a result, the history acts directly on search control and guides node selection by prioritizing candidate histories according to their estimated future search potential, allowing budget allocation to depend on each node's local trajectory and context (analyzed in Sec. 4.2). As illustrated in Fig. 1, compared to MAP-Elite sampling, which allocates budget based on predefined probabilities and may incur high randomness, ZetaEvolve reduces randomness by focusing on high-potential histories. Compared to PUCT, which may still favor locally high-scoring but saturated branches, ZetaEvolve downweights stagnated trajectories and prioritizes branches with remaining improvement potential, thereby capturing richer search signals and more effectively allocating limited search budget.

Experiments on multiple self-evolving benchmarks show that our approach achieves stronger convergence behavior, better average performance, and competitive or superior best-found solutions

[2]Code and experiment configurations: `https://github.com/Shenjiyang/Zetaevolve/tree/camera-ready`.

compared with representative baselines. These results suggest that smaller models, when combined with history-conditioned search control, can remain highly competitive even without expensive reinforcement learning updates. Our contributions are as follows:

- We model self-evolution from a node-centered POMDP perspective, combining local histories into self-evolving search. We then propose ZetaEvolve, a history-conditioned search strategy that maps node histories to potential values for adaptive node selection.
- We show that history-conditioned search control improves small-model parallel sampling without online parameter updates, yielding strong competitive performance across multiple benchmarks.

2 Preliminary

PUCT is a tree-search rule that balances exploitation and exploration when selecting which node or action to expand [20, 21]. At a search state s, PUCT typically selects the action

$$a^* = \arg\max_a \left(Q(s,a) + c_{\text{puct}}\, P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)} \right), \tag{1}$$

where $Q(s,a)$ is the current value estimate, $P(s,a)$ is a prior preference over actions, $N(s,a)$ is the visit count, and $c_{\text{puct}} > 0$ is a constant that controls the exploration strength. The first term favors actions with high estimated value, while the second term encourages exploration of promising but under-visited actions. In this way, PUCT allocates the search budget according to both current utility and an implicit notion of future potential.

POMDP [14, 23, 22] is defined by $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{O}, T, \Omega, r, \gamma)$, where $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{O}$ denote the latent state, action, and observation spaces, respectively, $T(s_{t+1} \mid s_t, a_t)$ is the transition kernel, $\Omega(o_{t+1} \mid s_{t+1}, a_t)$ is the observation model, $r(s_t, a_t)$ is the reward function, and $\gamma \in [0,1)$ is the discount factor. Under partial observability, the agent cannot directly access s_t, but instead acts based on the observable interaction history $h_t = (o_0, a_0, o_1, a_1, \ldots, a_{t-1}, o_t)$. In standard formulations, h_t induces a belief state $b_t(s) = \Pr(s_t = s \mid h_t)$, and policies are often written as $a_t \sim \pi(\cdot \mid b_t)$.

Instead of explicitly maintaining belief states, it is also standard to formulate the decision process directly in terms of observable histories, using a history-conditioned policy $a_t \sim \pi(\cdot \mid h_t)$. In this view, $V(h_t) = \mathbb{E}_\pi[\sum_{k=t}^{\infty} \gamma^{k-t} r_k \mid h_t]$ and $Q(h_t, a_t) = \mathbb{E}_\pi[\sum_{k=t}^{\infty} \gamma^{k-t} r_k \mid h_t, a_t]$, which satisfy

$$V(h_t) = \mathbb{E}_{a_t \sim \pi(\cdot \mid h_t)}[Q(h_t, a_t)]. \tag{2}$$

The recursion is $Q(h_t, a_t) = \mathbb{E}[r_t + \gamma V(h_{t+1}) \mid h_t, a_t]$, where $h_{t+1} = (h_t, a_t, o_{t+1})$. The standard POMDP objective is to maximize the expected discounted return $J(\pi) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, where the expectation is taken over trajectories induced by the policy and the initial state distribution.

3 Problem Modeling

3.1 Node-Centered Local POMDP View

We model self-evolution from a node-centered POMDP perspective. At search stage t, the current search tree induces a set of expandable nodes $\mathcal{U}_t$. Rather than treating the entire tree as a single POMDP rollout process, we associate each expandable node $u \in \mathcal{U}_t$ with a local partially observable decision process. Here t is a shared step index across all node-centered processes. At each step, selected nodes undergo local transitions, while unselected nodes retain their current state and history. Newly created nodes enter the tree with histories indexed by the step at which they are generated.

Latent State and Observable History. For each node $u \in \mathcal{U}_t$, let s_t^u denote its local latent search state at step t. We define s_t^u as the underlying structural gap and logical deficiency of the current implementation relative to the global optimum. Since this logical distance is not directly observable from the source code alone, we utilize a node-level observable history h_t^u as a proxy, which includes the current program, its context programs, the corresponding evaluation scores, and other local information around the node. We then formulate the subsequent value estimation and node selection directly in terms of histories. As illustrated in Fig. 2, h_t^u integrates both the ancestor path and the outcomes of already expanded child nodes. By treating history as a rich informative context, we

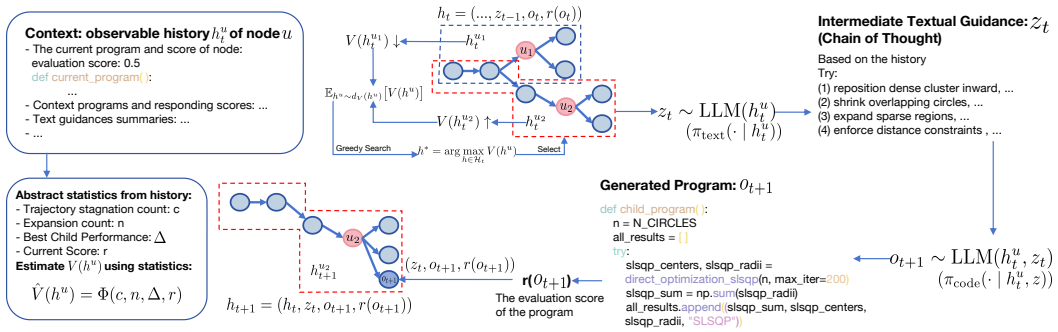


Figure 2: Self-evolving search under a POMDP view. At each step t, each node u in the search tree is associated with its own observable history h_t^u, denoted as dashed boxes. In self-evolution, previously generated children $o_{0:t}$ in the tree are also part of the history. After a selected node is expanded and new children o_{t+1} are evaluated, the history associated with that node changes accordingly. A typical way to use history is to compress prior code abstractions and evaluation outcomes into textual prompts for the LLM, which can be viewed as one form of history proxy.

can effectively approximate the unobservable state s_t^u and guide the search toward more promising directions with higher potential value.

Action and Transition. For a node u selected for expansion at step t, the rollout stage samples a packed expansion action $a_t = (z_t, o_{t+1}) \sim \pi(\cdot \mid h_t^u)$, where z_t denotes intermediate textual guidance and o_{t+1} denotes the generated child code. The action factorizes as $z_t \sim \pi_{\mathrm{text}}(\cdot \mid h_t^u)$ and $o_{t+1} \sim \pi_{\mathrm{code}}(\cdot \mid h_t^u, z_t)$, yielding $\pi(a_t \mid h_t^u) = \pi_{\mathrm{text}}(z_t \mid h_t^u)\pi_{\mathrm{code}}(o_{t+1} \mid h_t^u, z_t)$. Applying a_t induces a local latent-state transition $s_{t+1}^u \sim T(\cdot \mid s_t^u, a_t)$. Here o_{t+1} has a dual role: it is both part of the action and the observable program node associated with the updated local state. After evaluation, its score $r(o_{t+1})$, together with other local search outcomes, updates the history as $h_{t+1}^u = \mathrm{Update}(h_t^u, a_t, o_{t+1}, r(o_{t+1})) = \mathrm{Update}(h_t^u, z_t, o_{t+1}, r(o_{t+1}))$, where the second form follows from $a_t = (z_t, o_{t+1})$.

3.2 Objective of Self-Evolution

From the node-centered POMDP perspective, $V(h_t^u)$ characterizes the expected future return conditioned on the current history. In the context of self-evolving search, this is interpreted as the potential of node u to generate improvements beyond what has already been observed in its local history. A higher $V(h_t^u)$ indicates that the node remains promising for further expansion, while a lower value suggests that the local search around it is approaching saturation.

The goal of self-evolution, however, is not merely to improve local optima, but to allocate limited search budget over the whole tree to favor histories that are more likely to yield global improvement. This motivates a history-conditioned global objective defined over a traversal-induced distribution d_V of node histories,

$$J(\pi, d_V) = \mathbb{E}_{h_t^u \sim d_V(h_t^u)} \underbrace{\mathbb{E}_{a_t \sim \pi(\cdot|h_t^u)}\left[Q(h_t^u, a_t)\right]}_{V(h_t^u)}. \tag{3}$$

$d_V(h_t^u)$ denotes the distribution over selected expandable node histories induced by tree traversal. In most existing self-evolution methods [18, 4], the model is frozen during search, making $\pi(\cdot \mid h_t^u)$ fixed, $V(h_t^u)$ therefore becomes a deterministic function of the history, so optimizing $J(\pi, d_V)$ reduces to optimizing the traversal-induced history distribution d_V. Combining Eq. (3) with Eq. (2) yields

$$J^* = \max_{d_V} \mathbb{E}_{h^u \sim d_V(h^u)}\left[V(h^u)\right]. \tag{4}$$

This objective is applied at each search stage t, where decisions are made over the currently expandable node histories. From the search perspective, the self-evolution objective is therefore to allocate more budget to high-value histories across the tree.

4 ZetaEvolve

Since directly getting $V(h^u)$ is infeasible, a natural alternative would be to learn a neural estimator for $V(h^u)$ [20]. However, we do not adopt this design because self-evolving search is strongly distribution-shifting, rollout data are expensive, and each rollout irreversibly incorporates its outcomes into the node's observable history, making the original state unrecoverable for reuse. The problem is therefore less about learning a reusable value function and more about allocating search budget over the currently visible tree. This motivates us to seek a tractable heuristic for $V(h^u)$. To this end, we first consider PUCT as a natural baseline for approximating $V(h^u)$, as it allocates search budget based on node statistics and implicitly estimates the remaining sampling potential of each node under limited budgets. Building on this perspective, we propose ZetaEvolve, which improves upon PUCT by adapting its allocation strategy to better match the objective of self-evolution, yielding a more suitable approximation to $V(h^u)$.

4.1 Misalignment of Vanilla PUCT in Self-Evolving Search

PUCT is effective in search settings where the goal is to repeatedly sample actions to estimate their expected value under uncertainty. Yuksekgonul et al. [29] has adapted PUCT to self-evolving search by instantiating the prior term P with the score of the current program at a node, and the value term Q with the scores of its direct children. From this perspective, PUCT can be interpreted as a coarse proxy for estimating node potential under limited search budget, since its value and prior terms use sampling outcomes to guide whether a node should continue receiving search budget. Although PUCT leverages past sampling outcomes, it still estimates the value of repeatedly sampling a node $V(u)$ under limited budgets, rather than the value of the full history $V(h^u)$ represented by that node, including its previously generated children. As illustrated in Fig. 1, this leads to a mismatch with the objective of self-evolving search, which differs in three important ways.

First, self-evolution does not seek nodes with high expected or average child scores, but nodes that can still produce children better than those already observed. In POMDP perspective, generated children become part of the node's history. Therefore, if repeated expansions keep producing high average child scores but fail to improve the best child, the sampling value of that node's updated history should decrease. Vanilla PUCT does not explicitly suppress such nodes, and may therefore oversample locally saturated directions. *Second, vanilla PUCT does not model trajectory-level stagnation.* Under POMDP formulation of self-evolution, the evolutionary trajectory is part of the node's history, a node should be downweighted if the branch it lies on has not improved for a long time, even when its local score remains high. *Third, its sublinear exploration decay is mismatched to self-evolution.* The exploration term decays sublinearly with visit count, which is appropriate when the objective is to estimate a fixed expected value with increasing confidence, but not when the goal is to detect whether a node is still productive or already exhausted based on the history.

To formalize this intuition, consider a latent variable $Z_u \in \{0, 1\}$ for node u, where $Z_u = 1$ indicates that the node remains productive. Let $m_u^\star$ denote the best child score observed so far from node u, and assume $\Pr(Z_u = 1) = q_0,\ \Pr(Y > m_u^\star \mid Z_u = 1) = \theta,\ \Pr(Y > m_u^\star \mid Z_u = 0) = 0$. If the next f expansions from node u all fail to exceed $m_u^\star$, then Bayes' rule gives

$$\Pr(Z_u = 1 \mid f \text{ failures}) = \frac{q_0(1-\theta)^f}{(1-q_0) + q_0(1-\theta)^f}, \qquad \frac{\Pr(Z_u = 1 \mid f \text{ failures})}{\Pr(Z_u = 0 \mid f \text{ failures})} = \frac{q_0}{1-q_0}(1-\theta)^f. \quad (5)$$

Although $\Pr(Z_u = 1 \mid f \text{ failures})$ is not itself purely exponential in f, its posterior odds decay geometrically. This is precisely the behavior desired in self-evolving search: early tolerance to exploration, followed by sharp suppression of unproductive branches. Accordingly, in practice, we model the history-conditioned value as $V(h^u) \propto \rho^f$, an exponential decay elaborated in Sec. 4.2, where $0 < \rho < 1$ and f summarizes the accumulated evidence that node u is becoming unproductive.

4.2 Our Proposal: History-Conditioned Potential Value

Inspired by the POMDP framework and the above limitations of PUCT, we propose a potential value function $\hat{V}(h^u)$

$$\hat{V}(h^u) = r(u)\,\rho^{\frac{n_u + c_u}{\log\left(e + kN\right)}} + \beta\,\Delta_{\max}(u)\,\rho^{\frac{2n_u}{\log\left(e + kN\right)}}, \quad (6)$$

where $r(u)$, n_u, c_u, $\Delta_{\max}(u)$, and N are search-dependent statistics. Specifically, $r(u) > 0$ is the valid evaluation score of the program at node u, n_u tracks repeated expansions of node u without improvement, c_u measures how long the trajectory leading to u has gone without improvement (for details, see Appendix A.1), $\Delta_{\max}(u) \geq 0$ is the largest observed direct child improvement from u, and N denotes global search progress measured by the number of rollout steps, consistent with the stage index t. The parameters β, ρ, and k are hyperparameters, where $\beta > 0$ controls the relative weight of the child-improvement term, $k > 0$ modulates how global progress influences the decay rate, and $\rho \in (0, 1)$ controls how rapidly node contribution diminishes with repeated expansion and stagnation. We scale N logarithmically, drawing on the $\log t$ term in UCB [21] and logarithmic cooling in simulated annealing [8]. As $\log(e + kN)$ grows relative to the local penalty counts, penalties on untouched nodes relax while repeatedly expanded yet non-improving nodes remain suppressed. This reflects the intuition that heavily explored or stagnant regions are less likely to yield further improvements.

The first term provides a score-based prior for node u: a higher $r(u)$ indicates a higher initial likelihood of producing improvements. However, since self-evolution depends on whether further expansion can generate better children, this prior is discounted by the expansion count n_u and further penalized by the trajectory-level stagnation c_u, preventing high-score but non-improving nodes from dominating the search. The second term captures evidence of remaining improvement potential from observed children. A large $\Delta_{\max}(u)$ indicates that the node has produced meaningful improvements, while failure to further improve leads to exponential decay with n_u, gradually identifying exhausted nodes. Together, the first term acts as an initial prior, and the second term corrects it using observed child-improvement signals.

Formal Node Selection Strategy. We use $\hat{V}(h^u)$ as an estimate of the true history-conditioned value around node u. Since $\hat{V}(h^u)$ evolves with the search process, it provides a dynamic test-time mechanism for adapting node selection without updating the LLM parameters. Let $\mathcal{H}_t = \{h_t^u : u \in \mathcal{U}_t\}$ denote the set of observable histories associated with the candidate nodes at stage t. This potential provides a concrete operationalization of the traversal-induced history distribution d_V over $\mathcal{H}_t$. Under the frozen-policy objective in Eq. 4, the optimal node-selection distribution is

$$d_V^* = \arg\max_{d_V} \mathbb{E}_{h^u \sim d_V(h^u)} \big[\hat{V}(h^u)\big].$$

In the greedy case, this reduces to selecting the history with maximum value [28]:

$$h^* = \arg\max_{h^u \in \mathcal{H}_t} \hat{V}(h^u), \tag{7}$$

and returning the corresponding node u^* such that $h^{u^*} = h^*$. As illustrated in Fig. 2, this node-selection strategy selects nodes according to the potential values of their histories at the beginning of each rollout step.

Overall, ZetaEvolve proceeds as follows. At each stage, the algorithm computes a history-conditioned potential value for all candidate nodes, selects high-potential nodes, and expands them by sampling multiple candidate programs with the language model. The resulting candidates are evaluated and inserted into the search tree, updating node histories and search statistics. This process iteratively reallocates computation toward nodes with higher potential for further improvement. The full procedure is summarized in Algorithm 1.

5 Experiments

Setup. We evaluate our method on a diverse set of mathematical optimization and algorithmic design tasks, combining benchmarks from AlphaEvolve, GEPA [1], ShinkaEvolve, AdaEvolve, and ThetaEvolve. Specifically, our task suite includes three problems from ThetaEvolve: CirclePacking, ThirdAutoCorrIneq, and HadamardMatrix, together with eight additional tasks from AdaEvolve, comprising four mathematical optimization problems (Heilbronn (Triangles), Heilbronn (Convex), MinMax Distance, Signal Processing) and four ADRS systems tasks (EPLB, Prism, LLM-SQL, and TXN). This combined benchmark enables comprehensive evaluation across continuous optimization challenges and general algorithmic design scenarios. For details, see Appendix A.

Table 1: Results on four mathematical optimization tasks. Higher values are better (↑) across all tasks. ZetaEvolve is evaluated with the Distill-Qwen3-8B backbone for 60 steps.

	Heilbronn (Triangles) (↑)		Heilbronn (Convex) (↑)		MinMax Distance (↑)		Signal Processing (↑)	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Backbone: GPT-5								
OpenEvolve	.028 ± .006	.028	.025 ± .005	.027	.226 ± .003	N/A	.569 ± .047	.622
GEPA	.031 ± .002	.032	.025 ± .002	.027	.232 ± .120	.239	.689 ± .014	.705
ShinkaEvolve	.032 ± .012	.034	.023 ± .005	.026	.239 ± .001	.240	.485 ± .044	.533
AdaEvolve	.033 ± .001	**.036**	.027 ± .001	**.029**	.239 ± .009	**.240**	.698 ± .017	.718
Backbone: Gemini-3-Pro								
OpenEvolve	.033 ± .000	.035	.028 ± .009	.028	**.240 ± .000**	**.240**	.553 ± .015	.565
GEPA	.031 ± .002	.033	.023 ± .001	.027	.213 ± .008	.218	.617 ± .075	.680
ShinkaEvolve	.035 ± .001	**.036**	.028 ± .001	**.029**	**.240 ± .000**	**.240**	.480 ± .039	.505
AdaEvolve	**.036 ± .001**	**.036**	**.029 ± .000**	**.029**	**.240 ± .000**	**.240**	.593 ± .009	.603
Backbone: Distill-Qwen3-8B								
ZetaEvolve	.035 ± .002	**.036**	**.029 ± .001**	**.029**	**.240 ± .001**	**.240**	**.781 ± .004**	**.783**

Table 2: Comparison on four ADRS systems benchmark tasks. Higher is better (↑) for all reported tasks. Our method is evaluated with the Distill-Qwen3-8B backbone for 60 steps.

	EPLB (↑)		Prism (↑)		LLM-SQL (↑)		TXN (↑)	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Backbone: GPT-5								
OpenEvolve	.127 ± .000	.127	26.230 ± .000	26.230	.710 ± .010	.716	4239 ± 90	4329
GEPA	.134 ± .010	.145	26.190 ± .070	26.230	.713 ± .000	.713	3753 ± 204	3984
ShinkaEvolve	.118 ± .010	.127	26.260 ± .000	26.260	.712 ± .000	.713	4090 ± 338	4329
AdaEvolve	.134 ± .010	.145	26.300 ± .070	26.370	.746 ± .030	.775	4317 ± 29	4348
Backbone: Gemini-3-Pro								
OpenEvolve	.127 ± .000	.127	26.240 ± .010	26.240	.729 ± .010	.736	4109 ± 254	4274
GEPA	.127 ± .000	.127	26.160 ± .030	26.190	.713 ± .000	.713	3616 ± 481	4167
ShinkaEvolve	.120 ± .010	.127	26.250 ± .010	26.260	.721 ± .000	.721	3932 ± 343	4255
AdaEvolve	.145 ± .000	.145	26.260 ± .000	26.260	.743 ± .010	.752	4221 ± 89	4310
Backbone: Distill-Qwen3-8B								
ZetaEvolve	**.146 ± .001**	**.146**	**38.850 ± .320**	**39.220**	**.824 ± .137**	**.983**	**4427 ± 137**	**4525**

5.1 Main Results

Baseline Results against Large Language Model Self-Evolution Methods. We use a small model with large-scale parallel sampling, and apply history to node selection instead of placing long history in the prompt. Because current frontier methods are mostly API configurations, Tables 1–2 compare this setup with those systems as a cross-system comparison, to see its performance against the frontier. ZetaEvolve uses a local Distill-Qwen3-8B with large-scale parallel sampling, whereas the API baselines use stronger backbones and substantially smaller accepted-generation budgets. Under these regimes, ZetaEvolve is competitive on the mathematical tasks and obtains the highest reported scores on the four ADRS tasks. The Signal Processing run evolves a multi-stage adaptive filter with dynamic window selection, while LLM-SQL develops adaptive row-wise column reordering. On Prism, the high score comes from a repeatedly discovered low-coverage, high-quality regime favored by the published reciprocal-KVPR metric; Appendix F.8 verifies evaluator consistency and analyzes this quality–coverage trade-off. Controlled DeepSeek-V3, DeepSeek-V4-Flash, and Qwen3-Coder-Next comparisons are reported in Appendix F.4, together with explicit generation and evaluation budgets.

Baseline Results on Large-Scale Parallel Sampling Benchmarks. Prior comparisons varied in both model size and parallelism, we then evaluate ZetaEvolve under the same large-scale parallel sampling setup. We compare ZetaEvolve with the frozen model results of ThetaEvolve (MAP-Elite) in [27] and investigate the performance difference from the standard PUCT node selection method under identical settings. As shown in Tab. 3, our method achieves superior performance over ThetaEvolve and the vanilla PUCT approach on all three evaluated tasks. We further analyze the convergence behavior of the proposed method. Since the original work of ThetaEvolve does not

Table 3: Comparison among ZetaEvolve, ThetaEvolve, and Zeta$^-$+PUCT. ZetaEvolve and Zeta$^-$+PUCT use 60×512 rollouts, while the reported ThetaEvolve result uses 65×512. For ThirdAutoCorrIneq, lower is better (↓). Zeta$^-$+PUCT retains the other ZetaEvolve components and replaces only Eq. 6 with vanilla PUCT scoring.

	CirclePacking (↑)		HadamardMatrix (↑)		ThirdAutoCorrIneq (↓)	
Method	Avg	Best	Avg	Best	Avg	Best
Backbone: Distill-Qwen3-8B						
ThetaEvolve	2.6354195 ± N/A	2.6359831	0.5500 ± N/A	0.5733	1.5498 ± N/A	1.5084
Zeta$^-$+PUCT	2.6317538 ± .004	2.6359831	0.5655 ± .008	0.5714	1.5296 ± .010	1.5154
ZetaEvolve	**2.6359832 ± .000**	**2.6359834**	**0.5673 ± .013**	**0.5806**	**1.5134 ± .040**	**1.4731**

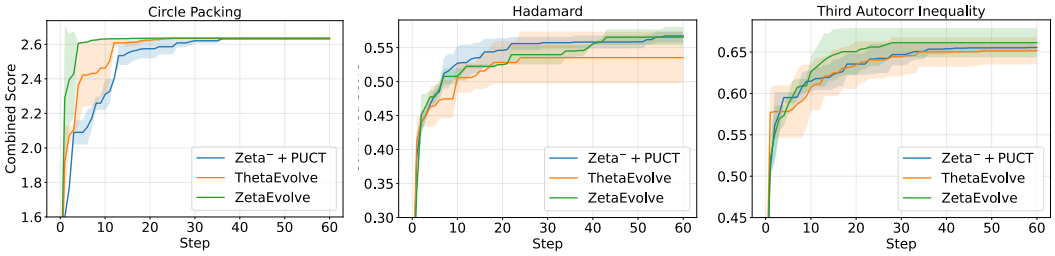


Figure 3: Comparison of convergence behavior among ZetaEvolve, ThetaEvolve, and Zeta$^-$+PUCT with Distill-Qwen3-8B for 60 steps. For ThirdAutoCorrIneq, the plotted quantity is the combined score, defined as the reciprocal of the original objective value; therefore, larger values indicate better performance in the plot.

provide convergence trajectories for its best reported runs, we re-implement and re-run ThetaEvolve under the same experimental configuration as ZetaEvolve for a fair comparison. Fig. 3 shows the convergence behavior of different methods. ThetaEvolve, corresponding to MAP-Elite style search, exhibits relatively larger variance, while Zeta$^-$+PUCT shows lower variance and more stable performance. Overall, ZetaEvolve achieves a more balanced convergence pattern, combining stable improvement with stronger final performance across tasks.

6 Case Study on ThirdAutoCorrIneq

Evolution across Steps. In Fig. 4, we visualize representative solutions on ThirdAutoCorrIneq from Step 0, Step 10, and Step 29. The objective decreases from 1.8418 to 1.5300 and further to 1.4731, indicating that self-evolution progressively discovers function constructions better aligned with the objective. Beyond numerical improvement, the evolution also leads to structural changes. As shown in the left panel, the learned function transitions from a highly irregular and noisy pattern at early stages to a smoother and more structured form at later stages. Correspondingly, the autoconvolution profile (right) evolves from a diffuse and unstructured shape to a more organized pattern with sharper variations and clearer peaks. These changes suggest that the search is not merely refining parameters, but actively reshaping the construction to better satisfy the underlying convolution objective. At the program level, the evolved solution also shifts from a relatively simple initialization-and-refinement procedure to a more structured optimization pipeline. We leave the detailed code-level comparison to Appendix E.

Comparison of Search Tree Structures. Figure 6 compares the depth–width distributions of search trees produced by ZetaEvolve, ThetaEvolve, and Zeta$^-$+PUCT, revealing how different rules allocate expansion budget. ThetaEvolve yields a substantially wider tree due to its MAP-Elite-style diversity objective and its default retention of all generated nodes, expanding broadly across depths. In contrast, Zeta$^-$+PUCT produces a deeper, depth-biased tree. As discussed in Sec. 4.1, PUCT may continue prioritizing nodes whose repeated expansions yield reasonable child scores but no longer improve the best observed children, causing search budget to concentrate on locally saturated branches. ZetaEvolve achieves a more balanced depth–width distribution, avoiding both excessive breadth and over-commitment to saturated trajectories, consistent with history-conditioned allocation.

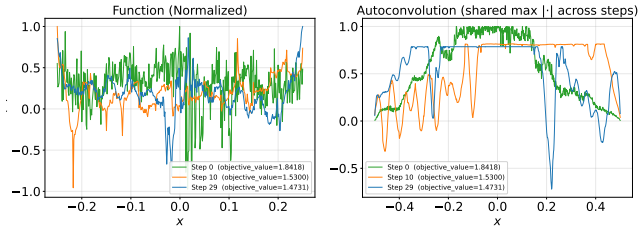


Figure 4: Comparison of representative solutions on ThirdAutoCorrIneq at different search steps with Distill-Qwen3-8B. **Left:** Normalized discrete $f(x)$ on $[-\frac{1}{4}, \frac{1}{4}]$, with each curve divided by $\max|f|$. **Right:** Autoconvolution $f * f$ under a shared scale $\max|\cdot|$ across steps.

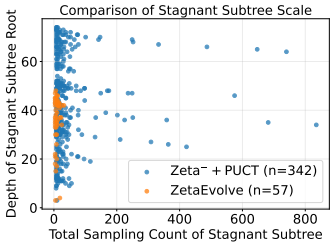


Figure 5: Comparison of subtree behavior under ZetaEvolve and PUCT with ProRL-1.5B-v2 for 100 steps.

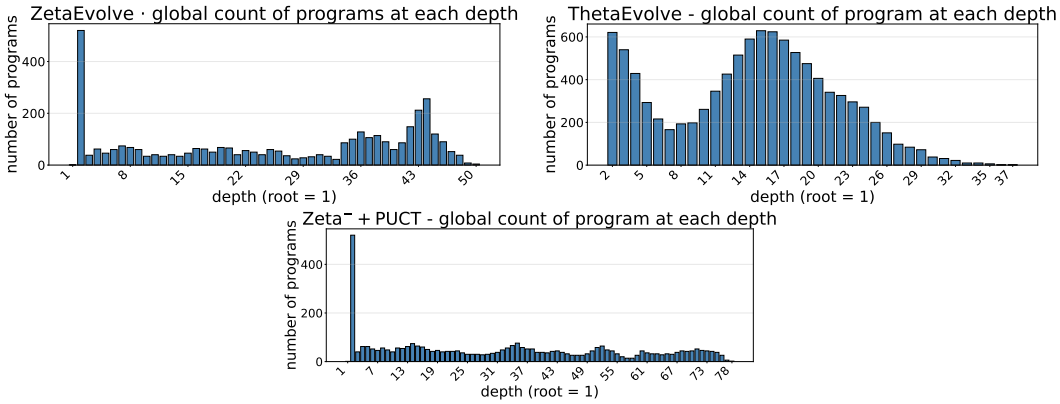


Figure 6: Depth and width analysis of different search tree structures with ProRL-1.5B-v2 for 100 steps.

We further examine this effect in Fig. 5, which compares stagnant subtree behavior under ZetaEvolve and Zeta$^-$+PUCT. For each subtree root, we measure descendant sampling counts and stagnant subtree depth, where stagnation indicates little or no improvement over the root. Zeta$^-$+PUCT produces more stagnant roots with larger sampling counts, while ZetaEvolve keeps most stagnant subtrees shallow with limited sampling. This indicates that ZetaEvolve allocates budget more selectively and avoids excessive expansion of unproductive branches.

7 Conclusion

In this paper, we revisit self-evolving search from a POMDP perspective and formulate it as a partially observable decision problem centered on node histories. Instead of treating history mainly as additional prompt context, we use node history as the basis of search control. This motivates ZetaEvolve, which maps observable histories to potential values for adaptive node selection and budget allocation.

Our goal is not to solve the induced POMDP exactly, but to use it as a modeling lens for explaining why history should guide search control. By incorporating trajectory-level signals such as stagnation and child improvement, ZetaEvolve better aligns search with the goal of finding further improvements rather than expanding saturated regions. Experiments show that this lightweight history-conditioned control improves convergence and final performance while supporting efficient parallel sampling with frozen models.

Limitations. First, we focus on search allocation without online reinforcement learning; integrating RL with history-conditioned search may further improve search policies. Second, ZetaEvolve mainly improves node selection, while better prompting, mutation operators, evaluators, and model adaptation remain complementary directions. Third, residual improvement potential is latent and approximated by hand-designed statistics; broader tests with other model classes, textual-history constructions, and longer-context models remain future work.

Acknowledgements and Disclosure of Funding

The authors are with Cambricon, DataCanvas, and the University of Southampton. The authors received no additional grant or third-party funding beyond those affiliations. The authors declare that they have no competing interests.

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[2] Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[4] Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

[5] Chentong Chen, Mengyuan Zhong, Ye Fan, Jialong Shi, and Jianyong Sun. Tide: Tuning-integrated dynamic evolution for llm-based automated heuristic design. *arXiv preprint arXiv:2601.21239*, 2026.

[6] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of machine learning research*, 24(240):1–113, 2023.

[7] Huan-ang Gao, Jiayi Geng, Wenyue Hua, Mengkang Hu, Xinzhe Juan, Hongzhang Liu, Shilong Liu, Jiahao Qiu, Xuan Qi, Yiran Wu, et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 1, 2025.

[8] Stuart Geman and Donald Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(6):721–741, 1984.

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

[10] Coleman Hooper, Sehoon Kim, Suhong Moon, Kerem Dilmen, Monishwaran Maheswaran, Nicholas Lee, Michael W Mahoney, Sophia Shao, Kurt Keutzer, and Amir Gholami. Ets: Efficient tree search for inference-time scaling. *arXiv preprint arXiv:2502.13575*, 2025.

[11] Jian Hu, Mingjie Liu, Shizhe Diao, Ximing Lu, Xin Dong, Pavlo Molchanov, Yejin Choi, Jan Kautz, and Yi Dong. Prorl v2: Prolonged training validates rl scaling laws. august 2025b. *URL https://hijkzzz. notion. site/prorl-v2*.

[12] Beichen Huang, Ran Cheng, Zhuozhao Li, Yaochu Jin, and Kay Chen Tan. Evox: A distributed gpu-accelerated framework for scalable evolutionary computation. *IEEE Transactions on Evolutionary Computation*, 2024.

[13] Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

[14] Leslie Pack Kaelbling, Michael L Littman, and Anthony R Cassandra. Planning and acting in partially observable stochastic domains. *Artificial intelligence*, 101(1-2):99–134, 1998.

[15] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[16] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

[17] Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

[18] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[19] Pretam Ray, Pratik Prabhanjan Brahma, Zicheng Liu, and Emad Barsoum. Adaptevolve: Improving efficiency of evolutionary ai agents through adaptive model selection. *arXiv preprint arXiv:2602.11931*, 2026.

[20] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[21] Aleksandrs Slivkins. Introduction to multi-armed bandits. *Foundations and Trends® in Machine Learning*, 12(1-2):1–286, 2019.

[22] Richard D Smallwood and Edward J Sondik. The optimal control of partially observable markov processes over a finite horizon. *Operations research*, 21(5):1071–1088, 1973.

[23] Edward Jay Sondik. *The optimal control of partially observable Markov processes*. Stanford University, 1971.

[24] Hongyuan Su, Yu Zheng, and Yong Li. Contextevolve: Multi-agent context compression for systems code optimization. *arXiv preprint arXiv:2602.02597*, 2026.

[25] Anja Surina, Amin Mansouri, Lars Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with llms: Evolutionary search meets reinforcement learning. *arXiv preprint arXiv:2504.05108*, 2025.

[26] Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. A survey on self-evolution of large language models. *arXiv preprint arXiv:2404.14387*, 2024.

[27] Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

[28] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[29] Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

[30] Yi Zhai, Zhiqiang Wei, Ruohan Li, Keyu Pan, Shuo Liu, Lu Zhang, Jianmin Ji, Wuyang Zhang, Yu Zhang, and Yanyong Zhang. \(x\)-evolve: Solution space evolution powered by large language models. *arXiv preprint arXiv:2508.07932*, 2025.

[31] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel Machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

[32] Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. HyperAgents. *arXiv preprint arXiv:2603.19461*, 2026.

[33] Jian Zhang, Zhiyuan Wang, Zhangqi Wang, Yu He, Haoran Luo, Lingling Zhang, Rui Mao, Qika Lin, Jun Liu, et al. Maxs: Meta-adaptive exploration with llm agents. *arXiv preprint arXiv:2601.09259*, 2026.

[34] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic Context Engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

[35] Shuo Zhang, Chaofa Yuan, Ryan Guo, Xiaomin Yu, Rui Xu, Zhangquan Chen, Zinuo Li, Zhi Yang, Shuhao Guan, Zhenheng Tang, et al. Evofsm: Controllable self-evolution for deep research with finite state machines. *arXiv preprint arXiv:2601.09465*, 2026.

A Experiment Details

A.1 Implementation Details

Besides the core history-conditioned potential $\hat{V}(h^u)$, we incorporate several complementary mechanisms to stabilize search dynamics and improve exploration efficiency.

1. Penalties for Revisit, Stagnation, and Child Improvement. For each node u, we maintain three statistics. The parent-reuse counter n_u^{raw} records repeated expansions of u: it is incremented when u is expanded without producing a strictly better direct child, and reset to 0 once such a child is found. The lineage stagnation depth c_u^{raw} records how long the trajectory leading to u has failed to improve its path-best score: for a newly retained child v, we set $c_v^{\mathrm{raw}} = 0$ if v improves the path-best score, and otherwise set $c_v^{\mathrm{raw}} = c_u^{\mathrm{raw}} + 1$. We then use shifted counters $n_u = \max(0, n_u^{\mathrm{raw}} - 1)$ and $c_u = \max(0, c_u^{\mathrm{raw}} - 1)$, so the first revisit or stagnation level is unpenalized.

The child-improvement signal is $\Delta_{\max}(u) = \max\{0, \max_{v \in \mathrm{Child}(u)}(r(v) - r(u))\}$. Thus, only positive direct child improvements contribute to the potential value; if u has no expanded children or no child improves over u, this term is 0.

The second term weights n_u by 2 because it omits c_u, so $2n_u$ decays like $n_u + c_u$ in the first exponent.

2. Sibling Blocking during Selection. Once a node is selected for expansion, its sibling nodes are temporarily excluded from selection in the current iteration. This avoids redundant exploration of highly similar subtrees and improves the diversity of explored branches within each iteration.

3. Ranking-Based Batch Selection. At each iteration, expandable nodes are ranked in descending order according to their potential $\hat{V}(h^u)$. We select a batch of B distinct high-potential parent nodes for expansion, so each parent node can be selected at most once within the same iteration. For each selected parent, we then sample M child programs, resulting in $B \times M$ rollouts per iteration.

4. Top-K Filtering for Database Maintenance. For each expanded parent node in an iteration, we retain only the top-K child programs among the candidates generated from that parent in the current iteration. These retained children are inserted into the search database and remain eligible for future selection, so a parent may accumulate children across multiple expansions. This controls memory growth per expansion, filters low-scoring candidates, and keeps the search tree compact while preserving high-quality descendants.

5. L-Length Trajectory Prompting for Model Input. When generating a new child program, the LLM is provided with the selected node's current program and score, together with the information of up to L ancestor nodes along its trajectory. For smaller models, L could be set to 0 or 1.

A.2 Experiment Setup

We follow the parallel infrastructure of ThetaEvolve and use batched sampling implemented with Slime. For the main experiments, we use the frozen DeepSeek-R1-0528-Qwen3-8B model [9] and run search for 60 steps. For hyperparameter grid search and search-tree-structure analysis, we use ProRL-1.5B-v2 [11] and run search for 100 steps. For these local-model parallel-sampling experiments, each iteration selects $B = 32$ distinct parent nodes, and samples $M = 16$ child programs from each selected parent, resulting in 512 rollouts per iteration on two H100 GPUs.

For the additional algorithmic mechanisms, we retain the top-K children per parent in the database with $K = 2$, and use a trajectory prompt window size of $L = 1$, meaning that the prompt contains the selected node together with its immediate parent. For the history-conditioned potential, we set $\beta = 1.5$, $k = 0.1$, and $\rho = 0.96$. They are shared across tasks and were fixed before the main runs. Tab. 5 checks sensitivity afterwards.

For comparisons against large-language-model baselines, we directly use the reported results from AdaEvolve [4]. In addition, for the DeepSeek-V3 mathematical-task comparison, we rerun AdaEvolve under the same hardware environment using its default project configuration. In this comparison, both ZetaEvolve and AdaEvolve use the same budget of 100 search iterations, with one node selected and one candidate program sampled per iteration.

For large-scale parallel-sampling baselines, the final ThetaEvolve results are taken from [27]. For convergence-behavior comparisons, however, we rerun ThetaEvolve under the same environment

and configuration as ZetaEvolve, so the resulting performance may differ from those reported in the original paper.

B Related Work

Evolutionary Program and Heuristic Discovery with Frozen LLMs A growing line of work treats large language models (LLMs) as *semantic mutation operators* inside evolutionary or population-based search loops, enabling automated discovery of programs, algorithms, and heuristics on tasks with executable or otherwise automatable evaluators. AlphaEvolve [18] popularized the "evolutionary coding agent" paradigm, demonstrating that iterative code editing guided by evaluator feedback can yield both scientific and engineering improvements. Subsequent open-source systems aim to replicate and extend this paradigm in a reproducible manner. ShinkaEvolve [16] focuses on sample efficiency via parent sampling, novelty-based rejection sampling, and bandit-driven model selection, while CodeEvolve [2] implements island-based genetic programming with an inspiration-based crossover operator and meta-prompting to explore solution space.

Beyond generic code evolution, recent works specialize for automated heuristic design. TIDE [5] highlights the coupling between discrete algorithmic structure and continuous numerical parameters, proposing a nested outer–inner loop that separates structural evolution from parameter tuning. In contrast, X-evolve [30] moves the evolutionary target from individual solutions to *solution spaces*, where an LLM generates parameterized programs that define a tunable subspace, followed by cheaper score-based search within that subspace.

Adaptive and Meta-Evolution for Efficiency and Robustness Although LLM-driven evolutionary systems can be effective, they are often token- and cost-intensive, sensitive to context limits, and prone to stagnation, motivating adaptive control of the search process itself. DeltaEvolve [13] reframes evolutionary agents under an EM view and replaces full-code history with structured *semantic deltas* and progressive disclosure, improving token efficiency while strengthening evolutionary guidance. For systems code optimization under API-only constraints, ContextEvolve [24] decomposes optimization context into semantic state, direction, and experience distribution managed by specialized agents, approximating RL-like sample efficiency without weight updates. ACE [34] instead evolves reusable context for a fixed model. Darwin Gödel Machine (DGM) [31] uses an archive to evolve agent implementations, while HyperAgents [32] extends this direction by evolving both agents and their improvement procedure. Their evolutionary targets are reusable context or the agent implementation itself; ZetaEvolve keeps the generator fixed and changes node selection within program search. DGM's archive selection is related, while ShinkaEvolve provides the more direct program-evolution baseline in our experiments.

A complementary direction is to make the *search strategy* adapt online. AdaEvolve [4] unifies local exploration intensity, global resource allocation across islands, and meta-guidance under an accumulated improvement signal. EvoX [17] goes further by *meta-evolving* the search strategy itself, periodically replacing selection/variation strategies based on progress and stagnation signals. Finally, in multi-LLM evolutionary pipelines, AdaptEvolve [19] explores uncertainty-driven model routing to reduce inference cost while preserving performance, though it requires token-level log-probabilities.

Test-Time Learning: Reinforcement Learning Inside Discovery Loops Pure search over frozen models cannot internalize newly discovered strategies. Several works therefore incorporate reinforcement learning (RL) during deployment or test time. ThetaEvolve [27] integrates test-time RL into an AlphaEvolve-like program-evolution pipeline, enabling open models to improve on open problems and suggesting partial transfer of "evolving capability" across tasks. Learning to Discover at Test Time (TTT-Discover) [29] formulates discovery as RL on a single test problem, explicitly optimizing for producing a single best solution (rather than average return) and demonstrating results across mathematics, kernel engineering, algorithm competitions, and biology.

A related perspective trains the LLM search operator using signals harvested from evolutionary exploration. EvoTune [25] alternates between evolutionary search (as exploration) and RL / preference optimization to update the LLM policy, improving discovery efficiency while emphasizing diversity preservation. These approaches connect naturally to broader advances in RL for reasoning models (e.g., DeepSeek-R1 [9] and prolonged RL training regimes such as ProRL v2 [11]), but they also

raise challenges in stability, reward design, and generalization to tasks with sparse or unverifiable rewards.

Self-Evolving Research Agents and Meta-Adaptive Reasoning Beyond evolving task-specific code, another thread studies *self-evolving agents* that can adapt their workflows and tool-use strategies. EvoFSM [35] improves controllability and stability by evolving an explicit finite-state-machine (FSM) workflow instead of unconstrained prompt rewriting, and by distilling successful and failure trajectories into a self-evolving memory. In multi-tool reasoning, MAXS [33] introduces lookahead-based meta-adaptive exploration and a trajectory convergence mechanism to mitigate local myopia and trajectory instability while controlling inference cost. Surveys of self-evolution [26, 7] provide a broader taxonomy of what/when/how to evolve (models, memory, tools, workflows) and highlight open problems in safety, evaluation, and scalable co-evolution.

Foundations and Infrastructure The above systems often build on classic decision-making and search principles. Multi-armed bandits [21] justify exploration–exploitation scheduling (e.g., UCB-style operator/prompt selection). POMDP theory and planning [14, 22, 23] formalize decision-making under partial observability, which can model discovery processes where only noisy outcome signals are observable. Modern test-time search also inherits ideas from tree search in games, exemplified by AlphaGo [20]. From an engineering standpoint, scalable evolutionary computation frameworks such as EvoX (distributed GPU-accelerated EC) [12] and efficient tree search mechanisms like ETS [10] provide practical infrastructure for large-scale experimentation.

C Algorithm

Algorithm 1 History-Conditioned Self-Evolving Search

Require: Initial program o^{u_0}, budget T, selected parents B, samples per parent M, retention size K, hyperparameters β, ρ, k

Initialize search tree with root node u_0 and program o^{u_0}
Evaluate o^{u_0}, construct $h_0^{u_0}$, and set $r^\star \leftarrow r(o^{u_0})$
Initialize statistics for u_0: $n_{u_0}^{\mathrm{raw}} \leftarrow 0$, $c_{u_0}^{\mathrm{raw}} \leftarrow 0$, $\Delta_{\max}(u_0) \leftarrow 0$
Initialize candidate node set $\mathcal{U}_0 \leftarrow \{u_0\}$
for $t = 0, 1, \ldots, T-1$ **do**
  Let $N \leftarrow t + 1$
  for each node $u \in \mathcal{U}_t$ **do**
    Refresh node history h_t^u from the current tree context
    Set $n_u \leftarrow \max(0, n_u^{\mathrm{raw}} - 1)$, $c_u \leftarrow \max(0, c_u^{\mathrm{raw}} - 1)$
    Compute

$$\hat{V}(h_t^u) = r(u)\rho^{\frac{n_u + c_u}{\log(e+kN)}} + \beta\Delta_{\max}(u)\rho^{\frac{2n_u}{\log(e+kN)}}.$$

  end for
  Sort $\mathcal{U}_t$ by $\hat{V}(h_t^u)$ in descending order
  Select top-B parents $\mathcal{B}_t \subseteq \mathcal{U}_t$ with sibling blocking and no repeated selection
  for each selected parent node $u \in \mathcal{B}_t$ **do**
    Initialize current candidate child set $\mathcal{C}_u^{(t)} \leftarrow \emptyset$
    for $m = 1, \ldots, M$ **do**
      Sample textual guidance $z_t^{(m)} \sim \pi_{\mathrm{text}}(\cdot \mid h_t^u)$
      Sample child program $o_{t+1}^{(m)} \sim \pi_{\mathrm{code}}(\cdot \mid h_t^u, z_t^{(m)})$
      Evaluate $o_{t+1}^{(m)}$ and obtain $r(o_{t+1}^{(m)})$
      Create child node v, assign $o^v \leftarrow o_{t+1}^{(m)}$, and construct h_{t+1}^v
      Initialize statistics for new node v: $n_v^{\mathrm{raw}} \leftarrow 0$, $c_v^{\mathrm{raw}} \leftarrow 0$, $\Delta_{\max}(v) \leftarrow 0$
      Add v to $\mathcal{C}_u^{(t)}$
      if $r(o^v) > r^\star$ **then**
        $r^\star \leftarrow r(o^v)$
      end if
    end for
    Retain top-K children $\mathcal{C}_{u,K}^{(t)} \subseteq \mathcal{C}_u^{(t)}$ ranked by evaluation score
    Insert $\mathcal{C}_{u,K}^{(t)}$ as children of u and refresh affected histories
    Update n_u^{raw}, $\Delta_{\max}(u)$, and c_v^{raw} for retained children $v \in \mathcal{C}_{u,K}^{(t)}$ following Appendix A.1
  end for
  Update $\mathcal{U}_{t+1}$ from all nodes eligible for future expansion
end for
return best score $r^\star$

D Details of Benchmark Tasks

D.1 Task Details for the ThetaEvolve Benchmarks

We evaluate on three tasks adopted from the ThetaEvolve benchmark: CirclePacking (Square), ThirdAutoCorrIneq, and HadamardMatrix. These tasks are all open mathematical optimization problems, but they differ substantially in search structure and evaluation behavior. In particular, two of them involve nontrivial implementation details in the verifier, which are important for interpreting the reported scores.

CirclePacking (Square). This task asks the optimizer to place a fixed number of pairwise disjoint circles inside a unit square so as to maximize the sum of their radii. In our experiments, we focus on the case $n = 26$, which has been widely used in prior self-evolving work. The task is highly

sensitive to geometric feasibility, since even a very small overlap or boundary violation invalidates a candidate configuration. For this reason, different prior works adopt slightly different numerical tolerances when checking feasibility, which in turn leads to slightly different optimization targets. ThetaEvolve distinguishes between a tolerant evaluation setting (following OpenEvolve) and a stricter zero-tolerance setting used in AlphaEvolve-style evaluation. Since the ThetaEvolve benchmark adopts the tolerant setting, we follow the same configuration in our main experiments for consistency.

More concretely, OpenEvolve evaluates packing constraints with a small absolute tolerance (e.g., `atol` $= 10^{-6}$), while other systems enforce stricter or even zero tolerance. As noted in ThetaEvolve, this seemingly minor implementation choice effectively changes the optimization problem, so results under different tolerances are not strictly comparable. When necessary, a solution found under the tolerant setting can be converted into a feasible strict solution by slightly shrinking the radii, but reported scores should still be interpreted with care across different works.

ThirdAutoCorrIneq. This task concerns the third autocorrelation inequality and aims to improve the upper bound on the corresponding constant by searching over structured candidate functions. Let C_3 denote the largest constant such that

$$\max_{-1/2\le t\le 1/2} |f * f(t)| \ge C_3 \left(\int_{-1/4}^{1/4} f(x)\,dx \right)^2$$

holds for every function $f : \mathbb{R} \to \mathbb{R}$. In practice, the search space is parameterized by a discretized step function, and each candidate is scored through a verifier based on its discrete autoconvolution values.

A key issue is that ThetaEvolve notes a mismatch between the theoretical inequality and the AlphaEvolve implementation. In AlphaEvolve, a step function is represented by a height sequence $\{h_i\}_{i=1}^n$, and its autoconvolution is evaluated at discrete points $\{\mathrm{conv3}(k)\}_{k=1}^n$. The upper bound computed in the AlphaEvolve implementation is

$$C_3^{(\mathrm{AlphaEvolve})} = \left| \frac{2n \max_k \mathrm{conv3}(k)}{\left(\sum_{i=1}^n h_i\right)^2} \right|,$$

whereas the natural discretization of the theoretical inequality gives

$$C_3^{(\mathrm{theory})} = \frac{2n \cdot \max_k |\mathrm{conv3}(k)|}{\left(\sum_{i=1}^n h_i\right)^2}.$$

Thus $C_3^{(\mathrm{theory})} \ge C_3^{(\mathrm{AlphaEvolve})}$, and the two formulas define closely related but distinct optimization objectives. Following ThetaEvolve, we use the corrected theoretical form $C_3^{(\mathrm{theory})}$ in our evaluator. Consequently, our scores are not directly comparable to the original AlphaEvolve-reported values under $C_3^{(\mathrm{AlphaEvolve})}$.

HadamardMatrix. This task is a version of the Hadamard maximal determinant problem. A Hadamard matrix is an $n \times n$ matrix H with entries in $\{\pm 1\}$ that attains the orthogonality condition

$$HH^\top = nI,$$

where I is the identity matrix. The corresponding maximal determinant problem asks for the largest possible value of $|\det(H)|$ over all $\{\pm 1\}$ matrices of order n, subject to the classical upper bound

$$|\det(H)| \le n^{n/2}.$$

In our experiments, we consider the case $n = 29$, following prior work. For this order, the best-known result (Orrick et al., 2003) achieves

$$|\det(H)| = 2^{28} \cdot 7^{12} \cdot 320.$$

Following previous work, we report the normalized score

$$s = \frac{|\det(H)|}{2^{28} \cdot 7^{12} \cdot 342}.$$

This normalization matches the evaluator used in our experiments and makes the task convenient for program search: the objective is to find a valid $\{\pm 1\}$ matrix with a large determinant under the normalized score. Compared with the previous tasks, HadamardMatrix is more explicitly combinatorial, since the search space consists of discrete sign patterns and small local changes can lead to large variations in determinant values.

D.2 Task Details for the AdaEvolve Benchmarks

Math. We evaluate on five mathematical optimization tasks from the AdaEvolve benchmark:

- **Circle Packing (Rectangle),** $N = 21$**.** Pack N disjoint circles inside a rectangle of perimeter 4 so as to maximize the sum of their radii.
- **Heilbronn Problem (Triangle),** $N = 11$**.** Place N points on or inside a triangle of unit area so that the minimum area of any triangle formed by the selected points is maximized.
- **Heilbronn Problem (Convex Region),** $N = 13$**.** Place N points on or inside a convex region of unit area so that the minimum area of any triangle formed by the selected points is maximized.
- **MinMax Distance,** $N = 14$**,** $d = 3$**.** Find 14 points in $\mathbb{R}^3$ that maximize the ratio between the minimum and maximum pairwise distances. We report the squared ratio $(d_{\min}/d_{\max})^2$, consistent with the evaluator.
- **Signal Processing.** Optimize a continuous signal-processing objective under noisy evaluation, where candidate programs transform an input signal and are scored by an external evaluator.

These five tasks cover both geometric construction problems and continuous optimization under noisy feedback.

ADRS systems tasks. We further evaluate on four ADRS systems tasks from the AdaEvolve benchmark:

- **Expert Parallelism Load Balancer (EPLB).** Balance expert-parallel load across GPUs by redistributing expert assignments, with the goal of reducing stragglers while preserving throughput.
- **Model Placement (Prism).** Optimize model-to-GPU placement cost by assigning model components to GPUs so as to minimize global execution cost under capacity constraints.
- **LLM-SQL.** Reorder tabular data for prefix efficiency, improving prefix-cache hit rates by reordering rows or columns without changing the underlying semantics.
- **Transaction Scheduling (TXN).** Minimize makespan in transaction execution by scheduling dependent transactions so as to reduce overall completion time.

Compared with the mathematical tasks, these ADRS tasks are more directly system-oriented and involve heterogeneous objectives, expensive evaluation, and strong sensitivity to scheduling and allocation decisions.

E Core Program Diff for ThirdAutoCorrIneq

E.1 Step 0 to 10

From stochastic local search to gradient-guided annealing.

```
 def construct_function():
     """
-    Construct an optimized discrete function on [-1/4, 1/4] that minimizes
-    the C3 autoconvolution constant using iterative refinement.
+    Construct an improved discrete function on [-1/4, 1/4] for C3 optimization
+    using simulated annealing and gradient-based refinement.
     """
```

```
     n = 400
     x = np.linspace(-0.25, 0.25, n)

-    # Start with a simple Gaussian-like envelope
-    center = 0.0
-    width = 0.1
-    heights = np.exp(-((x - center) ** 2) / (2 * width ** 2))
-
-    # Add oscillations for structure
-    heights += 0.3 * np.cos(2 * np.pi * 8.0 * x)
-    heights += 0.15 * np.sin(2 * np.pi * 16.0 * x)
-
-    target_sum = 25.0
-    heights = heights * (target_sum / np.sum(heights))
-    c3_value = compute_c3(heights)
-    best_sol = heights.copy()
-    best_score = c3_value
+    # Initialize function values with a chirp-like profile.
+    a = 1.0
+    initial_heights = np.cos(np.pi * a * x**2)
+
+    target_sum = 25.0
+    initial_heights = initial_heights * (target_sum / np.sum(initial_heights))
+    heights = initial_heights.copy()
+    best_heights = heights.copy()
+    current_c3, _ = compute_c3(heights)
+    best_c3_value = current_c3
```

Search update rule.

```
-    iterations = 20000
-    stages = [
-        (0, 0.25, 0.05),
-        (0.25, 0.5, 0.01),
-        (0.5, 1.0, 0.001),
-    ]
-
-    def try_additive_perturbation(func, step_size):
-        new_func = np.copy(func)
-        indices = np.random.choice(range(n), 10, replace=False)
-        for idx in indices:
-            new_func[idx] += (np.random.rand() - 0.5) * step_size
-        return new_func
-
-    def try_sign_flip_perturbation(func):
-        new_func = np.copy(func)
-        length = np.random.randint(1, 6)
-        start_index = np.random.randint(0, n - length + 1)
-        new_func[start_index:start_index+length] *= -1
-        return new_func
-
-    for i in range(iterations):
-        if np.random.rand() < 0.5:
-            perturbed_sol = try_additive_perturbation(best_sol, current_step_size)
-        else:
-            perturbed_sol = try_sign_flip_perturbation(best_sol)
-        new_score = compute_c3(perturbed_sol)
-        if new_score < best_score:
-            best_score = new_score
-            best_sol = perturbed_sol
+    T = 100.0
+    T_min = 0.01
+    cooling_rate = 0.999
+    gradient_step_size = 0.05
+    max_iterations = 5000
+    improvement_window = 500
```

```
+    stagnation_counter = 0
+
+    for i in range(max_iterations):
+        current_c3, conv_full = compute_c3(heights)
+        if current_c3 < best_c3_value:
+            best_c3_value = current_c3
+            best_heights = heights.copy()
+            stagnation_counter = 0
+        else:
+            stagnation_counter += 1
+
+        if stagnation_counter > improvement_window:
+            perturb_rand = np.random.uniform(0.5, 1.5, size=n)
+            new_heights = heights * perturb_rand
+            new_heights = new_heights * (target_sum / np.sum(new_heights))
+            new_c3, _ = compute_c3(new_heights)
+            if new_c3 < current_c3:
+                heights = new_heights
+                current_c3 = new_c3
+            stagnation_counter = 0
+
+        grad = compute_gradient(heights, conv_full, n)
+        candidate = heights + gradient_step_size * (-grad)
+        candidate = candidate * (target_sum / np.sum(candidate))
+        candidate_c3, _ = compute_c3(candidate)
+
+        if candidate_c3 < current_c3:
+            heights = candidate
+            current_c3 = candidate_c3
+            stagnation_counter = 0
+        elif np.random.random() < np.exp((current_c3 - candidate_c3) / T):
+            heights = candidate
+            current_c3 = candidate_c3
+            stagnation_counter = 0
+
+        T *= cooling_rate
+        gradient_step_size *= 0.9995
```

Objective interface and gradient signal.

```
 def compute_c3(heights):
-    n = len(heights)
-    len_conv = 2*n - 1
-    a_pad = fft.fft(heights, len_conv)
-    b_pad = fft.fft(heights, len_conv)
-    conv_full = fft.ifft(a_pad * b_pad)
-    conv_full = conv_full[:len_conv]
-    c3 = abs(2 * n * np.max(np.abs(conv_full)) / (np.sum(heights)**2))
-    return c3
+    n_val = len(heights)
+    conv_full = np.convolve(heights, heights, mode='full')
+    denomin = np.sum(heights)**2
+    c3 = 2 * n_val * np.max(conv_full) / denomin
+    return c3, conv_full
+
+def compute_gradient(heights, conv_full, n_val):
+    m0 = np.argmax(conv_full)
+    grad = np.zeros(n_val)
+    for i in range(n_val):
+        if m0 >= i and m0 - i < n_val:
+            grad[i] = 2 * heights[m0 - i]
+
+    # Project the gradient to preserve the fixed-sum constraint.
+    all_ones = np.ones(n_val)
+    component = np.dot(grad, all_ones) / n_val * all_ones
+    grad_projected = grad - component
```

```
+    return grad_projected
```

Explanation. The transition from Step 0 to Step 10 is the main algorithmic jump in this trajectory. Step 0 uses a hand-designed Gaussian envelope with sinusoidal perturbations, then performs a purely stochastic local search: each iteration proposes either a small additive perturbation or a short sign flip, and the program greedily keeps the candidate only if it improves the scalar C3 objective. This gives the initial program a simple exploration mechanism, but the search is not informed by the structure of the autoconvolution objective.

By Step 10, the program has reorganized the constructor around a different optimization view. The initialization changes to a chirp-like profile, and the objective function now returns both the C3 value and the full convolution. This extra convolution information is then used to build a gradient signal at the maximum-convolution location. The search loop combines three mechanisms: gradient descent steps under the fixed-sum normalization constraint, simulated-annealing acceptance of occasional worse candidates, and stagnation-triggered random rescaling to escape local minima. Thus, the program moves from black-box random perturbation toward a structured optimizer that explicitly uses information from the convolution landscape.

E.2 Step 10 to 29

Initialization and search schedule.

```
 def construct_function():
     n = 400
-    # Initialize function values (chirp-like function)
     x = np.linspace(-0.25, 0.25, n)
-    a = 1.0  # Chirp rate parameter
-    initial_heights = np.cos(np.pi * a * x**2)
+
+    # Initialize function values with a structured multi-peak profile.
+    window_start_index = n // 4
+    window_end_index = 3 * n // 4
+    window = np.zeros(n)
+    window[window_start_index:window_end_index] = 1.0
+
+    def gaussian(x, center, sigma):
+        return np.exp(-((x - center)**2) / (2 * sigma**2))
+
+    centers = np.array([-0.125, 0.0, 0.125])
+    sigma = 0.03
+    a = [1.0, 1.5, 1.0]
+
+    initial_heights = np.zeros(n)
+    for i, c in enumerate(centers):
+        initial_heights += a[i] * gaussian(x, c, sigma)
+
+    initial_heights = initial_heights * window / np.sum(window)

     target_sum = 25.0
     initial_heights = initial_heights * (target_sum / np.sum(initial_heights))
     heights = initial_heights.copy()
     best_heights = heights.copy()
     current_c3, _ = compute_c3(heights)
     best_c3_value = current_c3

-    T = 100.0
-    T_min = 0.01
-    cooling_rate = 0.999
-    gradient_step_size = 0.05
-    max_iterations = 5000
+    T = {0.4: 10000, 0.2: 1000, 0.1: 100, 0.01: 10, 0.001: 1}[0.4]
+    T_min = 0.001
+    cooling_stages = {'0.4': 5000, '0.2': 5000, '0.1': 5000,
+                      '0.01': 5000, '0.001': 5000}
```

```
+    cooling_rate = 3.0
+    gradient_step_size = {0.4: 0.05, 0.2: 0.03, 0.1: 0.01,
+                          0.01: 0.003, 0.001: 0.001}[0.4]
+    max_iterations = 100000
+    phase_duration = {0.4: 0, 0.2: 5000, 0.1: 10000,
+                      0.01: 15000, 0.001: 20000}
     improvement_window = 500
```

Convolution evaluation.

```
+import math
+
 def compute_c3(heights):
-    n_val = len(heights)
-    conv_full = np.convolve(heights, heights, mode='full')
-    denomin = np.sum(heights)**2
-    c3 = 2 * n_val * np.max(conv_full) / denomin
-    return c3, conv_full
+    n_val = len(heights)
+    s = np.sum(heights)
+    if s == 0:
+        return float('inf'), np.zeros(0)
+
+    conv_length = 2 * n_val - 1
+    next_power = 2**math.ceil(math.log2(max(conv_length, 1)))
+
+    padded_a = np.concatenate([heights, np.zeros(next_power - n_val)])
+    padded_b = np.concatenate([heights, np.zeros(next_power - n_val)])
+    fft_heights = np.fft.fft(padded_a)
+    fft_other = np.fft.fft(padded_b)
+    conv_fft = np.fft.ifft(fft_heights * fft_other)
+    conv_full = np.real(conv_fft[:conv_length])
+
+    c3 = 2 * n_val * np.max(conv_full) / (s**2)
+    return c3, conv_full
```

Gradient of the normalized objective.

```
 def compute_gradient(heights, conv_full, n_val):
-    m0 = np.argmax(conv_full)
-    grad = np.zeros(n_val)
-    for i in range(n_val):
-        if m0 >= i and m0 - i < n_val:
-            grad[i] = 2 * heights[m0 - i]
-
-    # Project the gradient to preserve the fixed-sum constraint.
-    all_ones = np.ones(n_val)
-    component = np.dot(grad, all_ones) / n_val * all_ones
-    grad_projected = grad - component
-    return grad_projected
+    s = np.sum(heights)
+    m0 = np.argmax(conv_full)
+    m = conv_full[m0]
+    base_term = (4 * n_val * m) / (s**3)
+
+    grad = np.zeros(n_val)
+    for i in range(n_val):
+        if 0 <= m0 - i < n_val:
+            j = m0 - i
+            if i != j:
+                d_m = 2 * heights[j]
+            else:
+                d_m = heights[j]
+        else:
+            d_m = 0.0
```

```
+
+        grad[i] = (2 * n_val / (s**2)) * d_m - base_term
+
+    return grad
```

Escape operators.

```
-def escape_strategy(heights, n_val):
-    """Apply one of two stochastic escape mechanisms."""
+def escape_strategy(heights, n_val, critical_idx=None):
+    """Apply one of three stochastic escape mechanisms,
+    optionally targeting critical points."""
     p = np.random.random()
     if p < 0.7:
+        if critical_idx is not None:
+            segment_length = min(n_val//5, 50)
+            center = critical_idx
+            start_index = max(0, center - segment_length//2)
+            end_index = min(n_val, center + segment_length//2)
+            segment = heights[start_index:end_index].copy()
+            segment *= 1.0 + np.random.uniform(-0.1, 0.1)
+            new_heights = np.concatenate(
+                [heights[:start_index], segment, heights[end_index:]]
+            )
+        else:
-        scale = 1.0 + np.random.uniform(-0.2, 0.2)
-        new_heights = heights * scale
+            scale = 1.0 + np.random.uniform(-0.1, 0.1)
+            new_heights = heights * scale
-    else:
+    elif p < 0.9:
         segment_length = np.random.randint(10, 60)
         start_index = np.random.randint(0, n_val - segment_length)
         new_heights = heights.copy()
         new_heights[start_index:start_index+segment_length] *= -1.0
+    else:
+        std = 0.05 * (25.0 / n_val)
+        noise = np.random.normal(0, std, n_val)
+        new_heights = heights + noise

     new_heights = new_heights * (25.0 / np.sum(new_heights))
     return new_heights
```

Explanation. The transition from Step 10 to Step 29 is less of a wholesale algorithm replacement and more of a refinement of the optimizer introduced at Step 10. The initialization is changed from a single chirp-like cosine profile to a structured, windowed, three-peak Gaussian profile. This gives the search a more localized and symmetric starting shape, with the central peak weighted more heavily than the two side peaks. At the same time, the search budget is greatly increased, and the annealing parameters are rewritten as staged dictionaries, indicating a move toward a longer multi-phase refinement process.

The objective evaluation is also made more scalable. Step 10 computes the full autoconvolution with direct convolution, while Step 29 switches to FFT-based convolution with explicit zero padding to the next power of two. This preserves the same objective form, but makes repeated evaluations cheaper for the longer search loop.

The most substantive mathematical refinement is in the gradient. Step 10 differentiates only the peak convolution term and then projects the resulting vector to respect the fixed-sum constraint. Step 29 instead differentiates the normalized objective

$$J(h) = \frac{2n \max(h * h)}{(\sum_i h_i)^2},$$

including the derivative of the denominator. This leads to the update

$$\frac{\partial J}{\partial h_i} = \frac{2n}{(\sum h)^2}\frac{\partial m}{\partial h_i} - \frac{4nm}{(\sum h)^3},$$

where m is the maximum convolution value. Thus, the gradient becomes more aligned with the actual score optimized by the evaluator, rather than only reducing the peak convolution in isolation. We note that the evolved gradient implementation is still approximate and not a fully exact derivative; for example, the diagonal case in the convolution derivative misses a factor of 2. Nevertheless, it captures useful local structure of the normalized objective and provides an effective directional signal for the subsequent search.

Finally, the escape logic is expanded from two untargeted stochastic operators to a richer interface that can optionally perturb neighborhoods around critical convolution indices and add localized noise. In this saved Step 29 program, the main loop still primarily uses stagnation-triggered random rescaling, but the added escape operators show that the evolved program is beginning to encode more specialized mechanisms for escaping poor regions of the convolution landscape.

F Additional Results

F.1 Ablation

We conduct an ablation study on the ThirdAutoCorrIneq task to evaluate the contribution of each component in ZetaEvolve. As shown in Tab. 4, removing any component leads to performance degradation compared to the full model. Disabling the Top-2 database filtering causes the largest drop, suggesting that database maintenance is important for filtering low-scoring children and preventing repeated sampling from many high-scoring but similar descendants of the same parent. Removing history trajectory prompting also degrades performance, indicating that ancestral context helps the model generate more coherent and progressively improved solutions. Disabling sibling blocking leads to a comparable degradation, suggesting that reducing redundant exploration among similar branches also improves search efficiency. Overall, these results show that each component contributes to the final performance, with Top-2 database maintenance having the strongest impact among the ablated mechanisms.

Table 4: Ablation study on ThirdAutoCorrIneq. Lower is better (↓). All variants are evaluated with the Distill-Qwen3-8B backbone for 60 steps.

Variant	ThirdAutoCorrIneq (↓)
Backbone: Distill-Qwen3-8B	
ZetaEvolve (Full)	**1.5134 ± 0.040**
w/o Top-2 DB	1.5334 ± 0.030
w/o History Trajectory Prompt	1.5272 ± 0.047
w/o Sibling Blocking	1.5309 ± 0.026

F.2 Hyper-parameter Grid Search

Tab. 5 reports a later sensitivity check on ThirdAutoCorrIneq with ProRL-1.5B-v2. Performance is sensitive to both k and ρ. The best single cell is $(k = 0.10, \rho = 0.94)$, at 1.666 ± 0.040. Across k, however, $\rho = 0.96$ has the best column mean (1.713) and, at $k = 0.10$, a much smaller variance (1.701 ± 0.003) than that best cell. Smaller k is generally preferable, and the largest ρ is not best, so overly persistent retention can hurt. This supports the shared main-run choice $\rho = 0.96$.

Table 5: Grid search for hyper-parameters k and ρ on ThirdAutoCorrIneq with ProRL-1.5B-v2 for 100 steps.

$k\backslash\rho$	0.90	0.92	0.94	0.96	0.98	Mean
0.10	1.738 ± 0.063	1.766 ± 0.151	$\mathbf{1.666 \pm 0.040}$	1.701 ± 0.003	1.721 ± 0.029	**1.718**
0.25	1.677 ± 0.027	1.756 ± 0.052	1.774 ± 0.089	1.693 ± 0.028	1.749 ± 0.034	1.730
0.50	1.728 ± 0.001	1.724 ± 0.077	1.750 ± 0.015	1.746 ± 0.025	1.763 ± 0.020	1.742
Mean	1.715	1.749	1.730	**1.713**	1.744	-

Tab. 6 is a second check on Circle Packing (Square) with DeepSeek-V4-Flash, 100 accepted generations and evaluations, and three seeds. All six cells are close; $\rho = 0.96$ has the highest mean across k.

Table 6: Grid search for k and ρ on Circle Packing (Square) with DeepSeek-V4-Flash. 100 accepted generations/evaluations, three seeds. Higher is better (↑).

k / ρ	0.90	0.94	0.96
0.1	2.6337 ± 0.0027	2.6307 ± 0.0036	2.6322 ± 0.0033
0.5	2.6321 ± 0.0033	2.6305 ± 0.0013	$\mathbf{2.6341 \pm 0.0026}$

F.3 Transfer to Large API-Based Models

Table 7: Supplementary results on mathematical tasks with the DeepSeek-V3 backbone for 100 search iterations. While the main paper emphasizes the advantages of ZetaEvolve under small-model parallel sampling, this experiment shows that ZetaEvolve also remains effective in a large API-based model setting. Higher is better (↑).

Method	CirclePacking (Square) (↑)	CirclePacking (Rect) (↑)	Heilbronn (Triangles)	Heilbronn (Convex)	MinMax Distance
Backbone: DeepSeek-V3					
ZetaEvolve	**2.6282 ± 0.0026**	**2.3658 ± 0.0001**	**0.0288 ± 0.0047**	**0.0288 ± 0.0004**	**0.2383 ± 0.0024**
AdaEvolve	2.6191 ± 0.0004	2.3511 ± 0.0122	0.0251 ± 0.0003	0.0270 ± 0.0032	0.2367 ± 0.0031

F.4 Matched-Backbone Evidence and Budget Accounting

Table 7 compares the two methods on five mathematical tasks with the same backbone and a budget of 100 search iterations per method. We further compare ZetaEvolve and AdaEvolve on two ADRS tasks using two additional model classes. All runs in Tab. 8 use identical backbones, instances, evaluators, environments, and 100-iteration budgets, corresponding to 100 accepted generations and 100 program evaluations per method. ZetaEvolve has a higher mean in all four settings, although the DeepSeek-V4-Flash margin on Prism is modest. Together with the Distill-Qwen3-8B experiments, these comparisons cover the evaluated distilled-reasoning, stronger API reasoning, and coder-specialized backbones; they do not establish model-agnostic performance beyond these classes.

Table 8: Matched comparisons on ADRS tasks over three seeds. Higher is better (↑).

Backbone	Method	EPLB	Prism
DeepSeek-V4-Flash	AdaEvolve	0.137 ± 0.008	26.43 ± 0.28
DeepSeek-V4-Flash	**ZetaEvolve**	$\mathbf{0.181 \pm 0.025}$	$\mathbf{26.80 \pm 0.79}$
Qwen3-Coder-Next	AdaEvolve	0.126 ± 0.001	30.25 ± 5.98
Qwen3-Coder-Next	**ZetaEvolve**	$\mathbf{0.136 \pm 0.005}$	$\mathbf{33.78 \pm 6.89}$

Tables 1–2 compare complete systems rather than controlled search rules. Table 9 makes the difference in operating regimes explicit. An accepted generation produces one program and one evaluation. For API methods, one accepted generation may require multiple raw calls because reflection, rejection sampling, or meta-guidance can introduce auxiliary calls. In the controlled local comparison, ZetaEvolve uses 30,720 rollouts and ThetaEvolve uses 33,280. In the matched API comparisons, both ZetaEvolve and AdaEvolve use 100 accepted generations and evaluations; raw calls can still differ because of method-specific auxiliary calls. On DeepSeek-V4-Flash Prism, ZetaEvolve averages 1.03 hours versus 3.49 hours for AdaEvolve, although runtime also reflects API latency and implementation overhead.

F.5 Controls against Simpler History-Aware PUCT

The Zeta$^-$+PUCT comparison in Tab. 3 is selection-only: it retains the backbone, prompts, top-K retention, sibling blocking, ranked batch selection, evaluator, and rollout budget, replacing only Eq. 6 with vanilla PUCT. Table 11 separately tests whether adding a single expansion- or stagnation-aware decay to PUCT explains the gain. The simple stagnation-aware variant is both worse and less stable than vanilla PUCT, while the full history-conditioned potential performs best. Thus, generic stagnation awareness alone does not account for the improvement; Tab. 10 provides complementary within-method component ablations.

Table 9: Generation and evaluation budgets for the cross-system comparisons. NR denotes not reported.

Method	Backbone	Iter.	Accepted/iter.	Gen./eval.	Raw calls	Compute
ZetaEvolve	Distill-Qwen3-8B	60	512	30,720/30,720	30,720	2×H100
OpenEvolve	GPT-5/Gemini-3-Pro	100	1	100/100	NR	NR
GEPA	GPT-5/Gemini-3-Pro	100	1	100/100	NR	NR
ShinkaEvolve	GPT-5/Gemini-3-Pro	100	1	100/100	NR	NR
AdaEvolve	GPT-5/Gemini-3-Pro	100	1	100/100	$\geq$ 100, aux. NR	NR

Table 10: Ablation of components in the history-conditioned potential value (Eq. 6) on Circle Packing with the DeepSeek API (100 LLM calls). Higher is better (↑).

Variant	Circle Packing (Square) (↑)
ZetaEvolve (Full)	$\mathbf{2.6282 \pm 0.0026}$
w/o decay on $r(u)$ term (no visit/stagnation-based multiplicative decay)	2.5904 ± 0.0322
w/o $\Delta_{\max}$ term (the second term)	2.6205 ± 0.0053
w/o c_u in decay exponent of $r(u)$ term (n_u only)	2.6011 ± 0.0372
replace $\rho^{x/D}$ by PUCT-style sublinearity decay on both terms	2.6090 ± 0.0318

F.6 Sensitivity to Textual History Length

The trajectory length L controls only the ancestor programs placed in the generation prompt. It is distinct from the structured statistics n_u, c_u, and $\Delta_{\max}(u)$, which summarize expansion outcomes for node selection. Table 12 shows a sensitivity check on Heilbronn (Triangles). Including one ancestor gives a modest improvement in this setting, while longer textual histories do not improve performance. We therefore use $L = 1$ in the main experiments. More systematic history construction and tests with stronger long-context models remain future work.

F.7 Predictive Value of History Statistics

We replay 24 runs across Circle Packing, EPLB, and Prism. Each pre-expansion event is one record, and a node may reappear after its history has changed. For node u expanded at iteration t, let

$$G_{t,u} = \left(M_u^{\text{future}} - M_t\right)_+, \tag{8}$$

where M_t is the incumbent before expansion and M_u^{future} is the best score subsequently reached in the descendant subtree of u. This target credits delayed descendant improvements rather than only the direct child generated at t.

Table 13 reports Spearman correlation with $G_{t,u}$. Current score adds information beyond global progress, and the expansion and lineage statistics raise the correlation from 0.231 to 0.367. We also define a binary target indicating whether the subtree ever exceeds the incumbent after the expansion. Within each run, records are divided into early, middle, and late thirds by iteration rank. Adding n_u and c_u improves AUROC in every stage, so their predictive value is not only a proxy for global search progress. This analysis supports the use of observable history statistics as predictors of residual productivity; it does not imply that Eq. 6 is the unique solution to the POMDP formulation.

F.8 Prism Evaluator Verification and Solution Distribution

Prism uses 50 evaluation instances and the published score

$$\text{Score} = 1/\operatorname{mean}(\text{KVPR}_{\text{successful}}) + \text{success rate}, \tag{9}$$

Table 11: Matched controls on Circle Packing (Square) with DeepSeek-V4-Flash, 100 accepted generations/evaluations, and three seeds. Higher is better (↑); $D = \log(e + kN)$.

Variant	Selection score	Score
ZetaEvolve (Full)	Eq. 6	$\mathbf{2.6341 \pm 0.0026}$
Vanilla PUCT	$Q + U_{\text{PUCT}}$	2.6278 ± 0.0073
PUCT + Q expansion decay	$Q\rho^{2n_u/D} + U_{\text{PUCT}}$	2.6269 ± 0.0123
PUCT + stagnation decay	$Q + U_{\text{PUCT}}\rho^{c_u/D}$	2.5545 ± 0.1370

Table 12: Sensitivity to textual trajectory length L on Heilbronn (Triangles) with DeepSeek-V4-Flash. Higher is better (↑).

L	0	1	2	3
Score	0.03178	**0.03253**	0.03140	0.02901

Table 13: Predictive value of history statistics over 24 replayed runs.

Predictors	Spearman
N	0.134
$N + r$	0.231
$N + r + n$	0.274
$N + r + n + c$	**0.367**

Stage	Positive rate	$N + r$	$N + r + n + c$
Early	0.66	0.690	**0.748**
Middle	0.24	0.509	**0.636**
Late	0.07	0.586	**0.684**

where success rate is the fraction of instances accepted as successful. Our strict evaluator leaves this objective and its normalization unchanged while enforcing valid IDs, complete one-to-one assignment, per-GPU memory usage no greater than 80 GB, immutable inputs, and process-level timeouts. The same strict evaluator is used by both methods in the matched experiments. Table 14 shows that every valid program receives the same numerical score under both evaluators. The invalid Qwen AdaEvolve candidate is rejected for an incomplete assignment, while the high-scoring Distill-Qwen3-8B ZetaEvolve program receives 39.22 under both evaluators. The gain is therefore not caused by an evaluator mismatch.

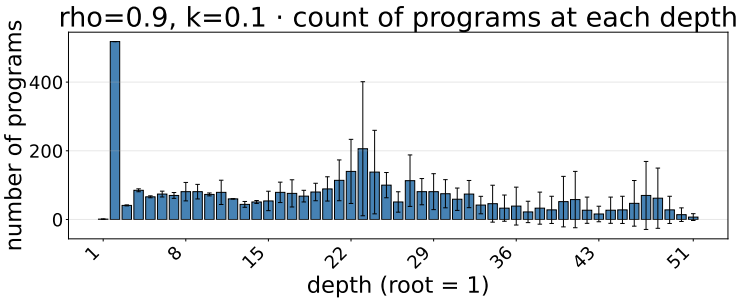
Table 14: Cross-verifier comparison for representative Prism programs.

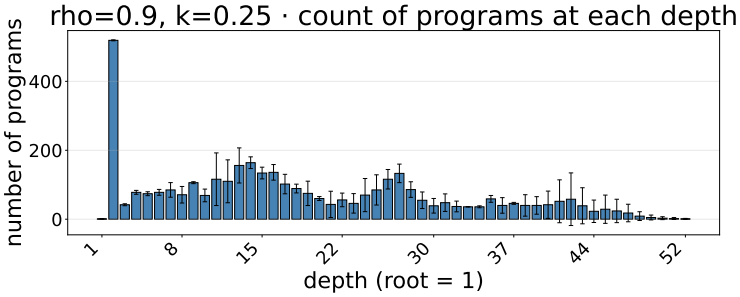
Candidate	Baseline	Strict	Diagnosis
Qwen Ada, s3408	94.95	0.00	Incomplete assignment
V4 Ada, s3409	26.47	26.47	Valid, identical
Qwen Zeta, s3407	42.90	42.90	Valid, identical
Qwen Zeta, s3408	26.25	26.25	Valid, identical
Distill-Qwen3-8B Zeta, s4408	39.22	39.22	Valid, identical

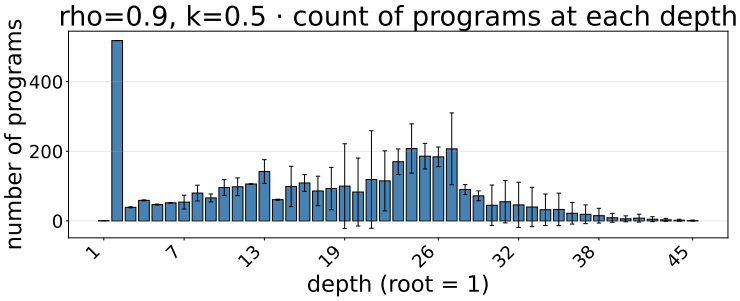
Table 15: Score distribution over 2,991 retained programs.

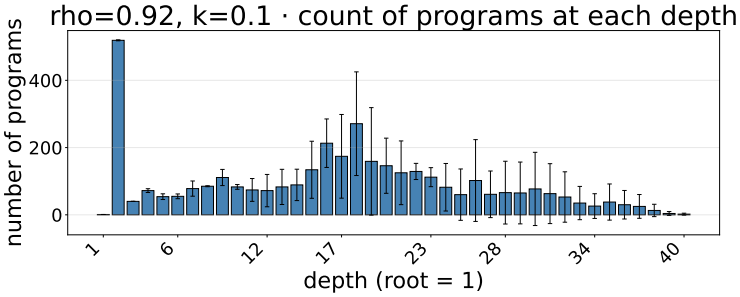
Success rate	Share	Mean	Max
≈ 0	3.1%	0.00	0.00
0.02–0.04	6.3%	34.07	39.22
0.05–0.10	2.4%	26.24	28.91
0.10–0.20	1.9%	26.17	26.55
0.20–0.50	0.4%	23.27	25.29
0.50–0.90	0.7%	22.88	26.80
0.90–1.00	85.1%	23.13	27.09

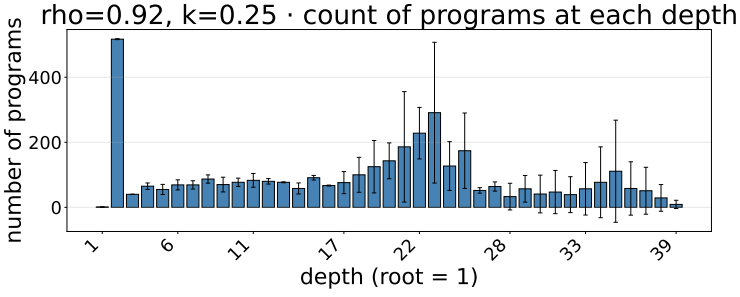
The high Prism score reflects a quality–coverage trade-off in the published metric. For s4408, the reciprocal-KVPR term is 39.20 while the success rate is 0.02, so the first term dominates the combined score. To determine whether this is an isolated solution, we analyze all 2,991 programs retained in the Distill-Qwen3-8B database. As shown in Tab. 15, most candidates retain near-complete coverage, while a smaller low-coverage regime receives higher scores. In total, 164 programs (5.48%) score above 30, all in the 0.02–0.04 success-rate band. Unique program hashes and distinct parent nodes account for 41% and 50% of this set, respectively. Thus, the search repeatedly discovers the same metric-level trade-off through multiple programs and branches, rather than relying on one lucky program; the improvement should nevertheless be interpreted as optimizing the published quality–coverage objective, not as uniformly increasing coverage.

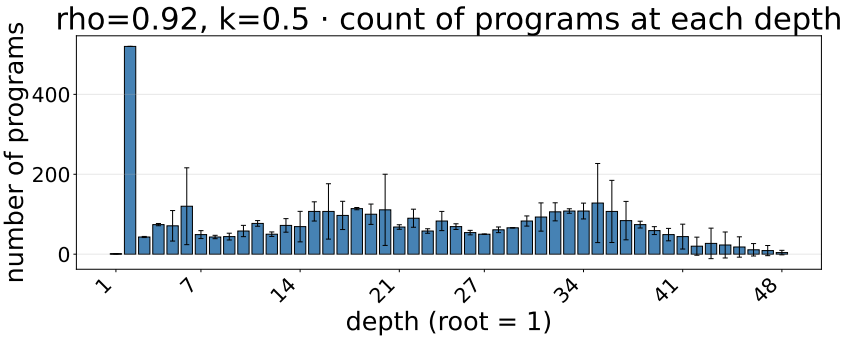

rho=0.9, k=0.1 · count of programs at each depth
number of programs
0
200
400
1
8
15
22
29
36
43
51
depth (root = 1)

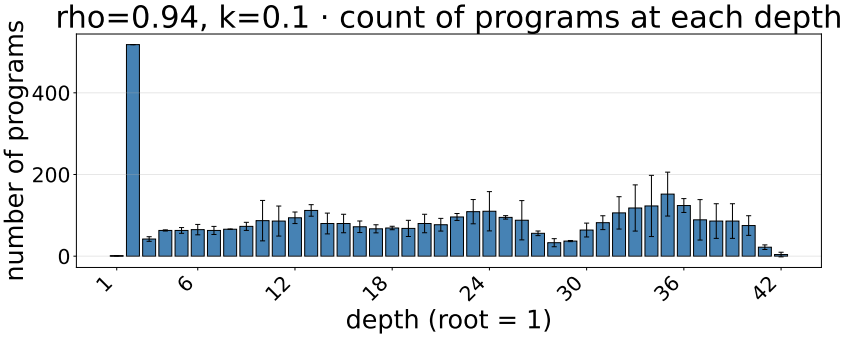

rho=0.9, k=0.25 · count of programs at each depth
number of programs
0
200
400
1
8
15
22
30
37
44
52
depth (root = 1)

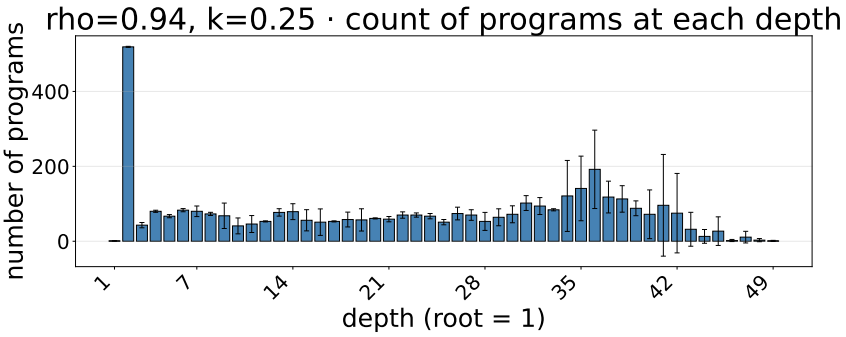

rho=0.9, k=0.5 · count of programs at each depth
number of programs
0
200
400
1
7
13
19
26
32
38
45
depth (root = 1)

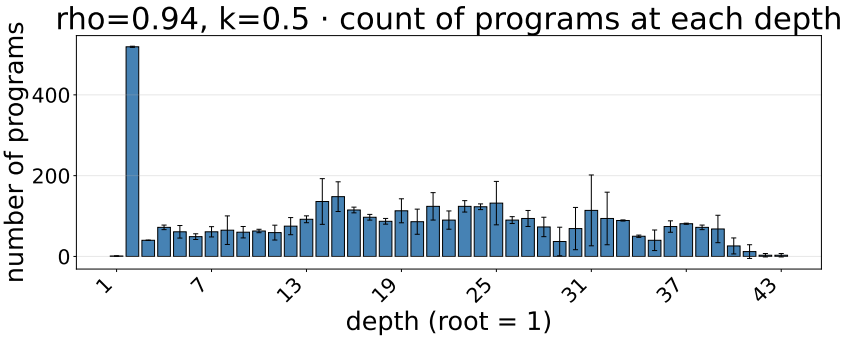

rho=0.92, k=0.1 · count of programs at each depth
number of programs
0
200
400
1
6
12
17
23
28
34
40
depth (root = 1)

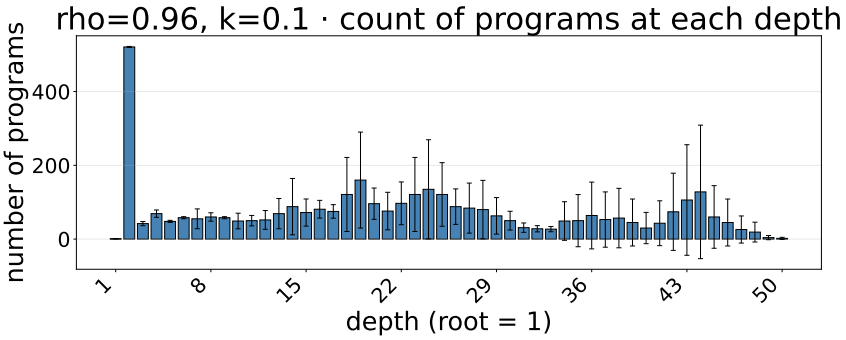

rho=0.92, k=0.25 · count of programs at each depth
number of programs
0
200
400
1
6
11
17
22
28
33
39
depth (root = 1)


rho=0.92, k=0.5 · count of programs at each depth
number of programs
0
200
400
1
7
14
21
27
34
41
48
depth (root = 1)


rho=0.94, k=0.1 · count of programs at each depth
number of programs
0
200
400
1
6
12
18
24
30
36
42
depth (root = 1)


rho=0.94, k=0.25 · count of programs at each depth
number of programs
0
200
400
1
7
14
21
28
35
42
49
depth (root = 1)


rho=0.94, k=0.5 · count of programs at each depth
number of programs
0
200
400
1
7
13
19
25
31
37
43
depth (root = 1)


rho=0.96, k=0.1 · count of programs at each depth
number of programs
0
200
400
1
8
15
22
29
36
43
50
depth (root = 1)

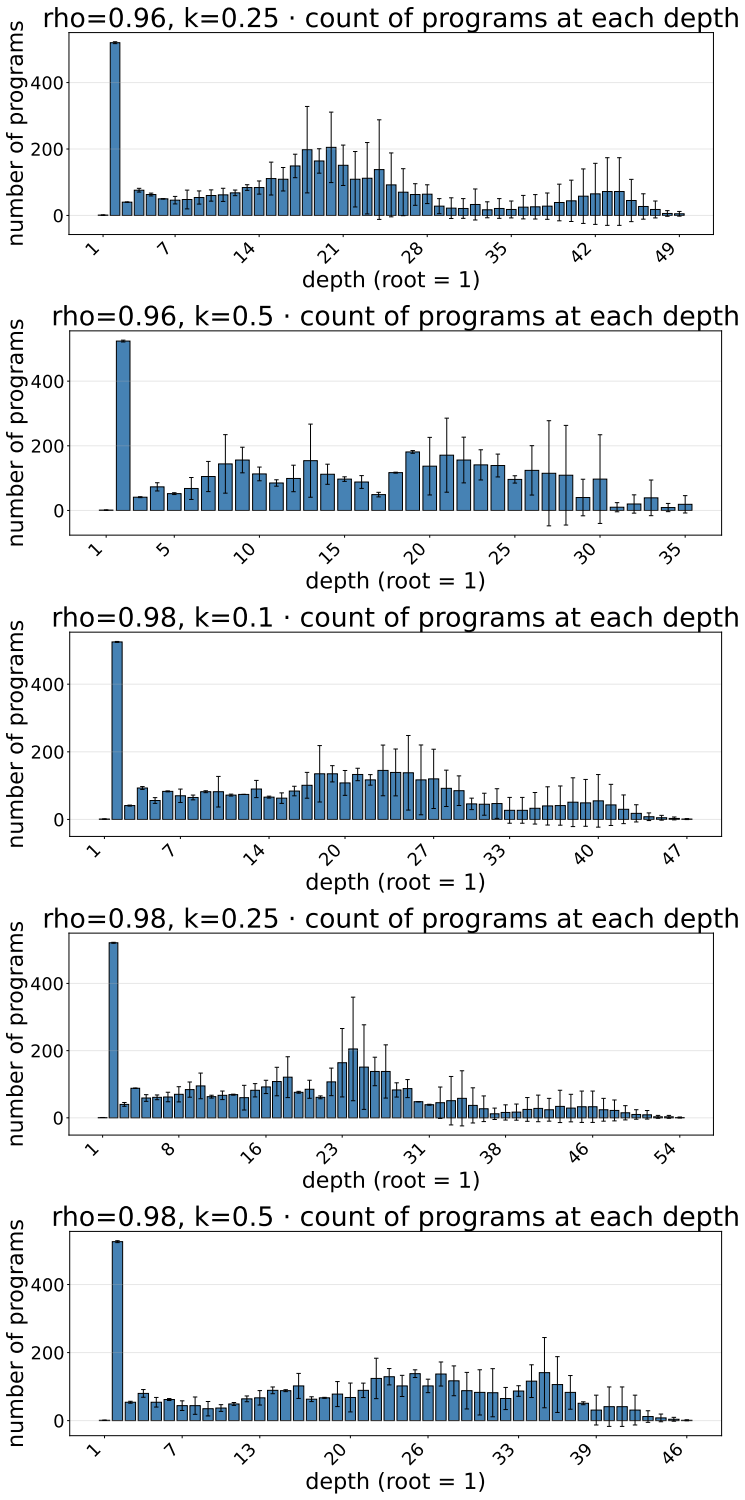


Figure 7: Per-depth width of the search tree under ZetaEvolve with ProRL-1.5B-v2 for 100 steps. The number of programs is unevenly distributed across depths and exhibits a multi-peak, wave-like pattern rather than a uniform shape. Instead of expanding the tree monotonically or uniformly across depths, ZetaEvolve allocates search budget selectively, producing several local width peaks at different depths and a thinner tail in deeper regions.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The claims made in the abstract and introduction are empirically verified in Sec. 5

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: We discuss the limitation in Sec. 7

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: We explained the full computation process of our formula in order to help everyone better understand the value of this method and provide some theoretical basis for the experimental results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: We provide detailed experimental configurations and report averaged results across three independent random seeds to ensure the reproducibility of our main findings.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: We will provide data, code, and instructions for the final paper.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: We provide experimental details in Sec. 5

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: We report averaged results across three independent random seeds to ensure the reproducibility of our main findings.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We provide information about the computer resources used

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have read and understood the code of ethics and have made every effort to adhere to it.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: There is no societal impact of the work performed.

    Guidelines:

    - The answer NA means that there is no societal impact of the work performed.
    - If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: We design a new algorithm instead of releasing any new models or datasets

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We use the datasets and models from AdaEvolve and ThetaEvolve, both of which are open-sourced projects and are properly cited in the paper.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: We will release our codebase along with detailed README files. We implement the SLIME parallel sampling framework by referring to ThetaEvolve, while all other algorithmic components and code are independently developed by ourselves.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: This work does not use crowdsourcing or human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: This work does not use crowdsourcing or human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: We used an LLM solely for editing assistance.

    Guidelines:

    - The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.