SELF-EVOLVING ALGORITHM-DESIGN AGENTS: ESCAPING IN-CONTEXT EVOLUTIONARY STAGNATION VIA POPULATION-CURATED POLICY OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models are increasingly participating in complex real-world tasks in the form of algorithm-design agents, designing and refining algorithms. Many successful algorithm-design agents adopt pure in-context evolutionary frameworks, but they may quickly plateau in domains that require specialized knowledge. Parametric adaptation offers a way to internalize specialized knowledge, but conventional training requires abundant domain-specific corpora while high-quality algorithms are scarce in complex algorithm-design scenarios. In this paper, we propose sample-efficient parametric self-evolution where agents can explore and learn from self-generated algorithms. First, we characterize in-context evolutionary stagnation and analytically propose the Improvement Chain proposition, showing how learning successive self-generated algorithms can locally increase the likelihood of neighboring algorithms. Motivated by this local-transfer perspective, we further propose Population-Curated Policy Optimization (PCPO) to utilize a global population and a hybrid policy update scheme for retaining and reusing high-quality, diverse self-generated algorithms, shifting the policy towards stronger algorithms. In the task of learning rate schedule design for global placement in electronic design automation, trained only on 4 chip cases, PCPO outperforms the state-of-the-art in-context evolutionary methods (e.g., OpenEvolve and ShinkaEvolve) on average across 16 chip cases. With an 8B-size base model, PCPO achieves competitive performance compared to frontier closed-source models such as GPT-5.5. PCPO also reduces inference-time token cost by internalizing grounded domain knowledge and prompt distillation. Moreover, PCPO achieves significant speedups on four GPU kernel designs, with an average of 8.27× speedup against the PyTorch Eager baseline. Our code is available at `https://anonymous.4open.science/r/PCPO-222D/`.

1 INTRODUCTION

As the capabilities of large language models (LLMs) continue to grow, LLMs are increasingly participating in complex real-world tasks in the form of agents (Wang et al., 2024; Xi et al., 2025). The self-evolving algorithm-design agent (Blount et al., 2025) is one promising form for completing complex real-world tasks, where the agent solves them from a high-level decision-making perspective by iteratively designing and refining algorithms, achieving outstanding results in scenarios like classical combinatorial optimization problems and open mathematical discoveries with pure in-context evolutionary frameworks (Liu et al., 2024a; Novikov et al., 2025; Lange et al., 2026).

However, in complex scenarios that require specialized knowledge, pure in-context evolution tends to quickly plateau (Liu et al., 2025a; Lu et al., 2026; Lange et al., 2026), whose diagnostic experiments are shown in Figure 2, posing obstacles to constructing effective self-evolving algorithm-design agents. A natural way to cope with specialized knowledge is to embed abundant domain-specific corpora in the model weights with continual pre-training (CPT) or supervised fine-tuning (SFT) (Liu et al., 2025b; Guo et al., 2025b; Ke et al., 2025). However, algorithm-design scenarios such as global placement (Chu, 2009) in electronic design automation (EDA) and GPU kernel optimization (Ouyang et al., 2025) offer insufficient labeled, high-quality algorithms. Therefore, self-evolving algorithm-design agents that automatically explore, enhance, and parametrically learn

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

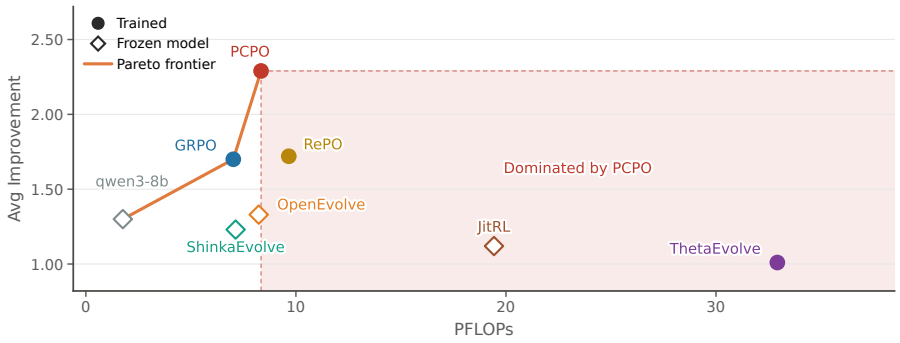


Figure 1: Average online improvement on 4 chips against total training and inference cost. PCPO achieves the largest improvement within acceptable computational cost, and the trained model can generalize to other chip cases with minimal inference-time token cost as shown in Section 6.1. Frozen-model methods underperform the base model due to the stagnation described in Section 4.1 and incur substantial computational cost due to repeated generation refinement and long thinking.

from self-generated algorithms demonstrate great significance. Recently, some works have made attempts to enhance evolution with reinforcement learning (RL) at inference time (Wang et al., 2026; Yuksekgonul et al., 2026), but they still face the typical sample-efficiency challenge of RL and do not make full use of the guidance from high-quality and diverse self-generated algorithms, leaving the potential benefits of population-level experience reuse across policy updates underexplored.

Therefore, in this work, we develop self-evolving algorithm-design agents that evolve via exploring and training on self-generated algorithms. We first characterize in-context evolutionary stagnation through diagnostic experiments. The observed sensitivity to domain information and the repeated low-quality generations motivate the hypothesis that hallucinated self-generated context can reinforce a cycle of biased outputs and local stagnation. Then we introduce the Improvement Chain as a motivating abstraction for studying local probability transfer between neighboring algorithms, with a conditional proposition that provides a conceptual account of how learning from curated intermediate algorithms may locally shift the task-conditional distribution toward neighboring algorithms. Motivated by this local-transfer perspective, we further propose Population-Curated Policy Optimization (PCPO) to utilize a global population for retaining and reusing high-quality, diverse self-generated algorithms as promising candidates. PCPO also adopts a hybrid policy update scheme, where refreshing on-policy updates enhance exploration and fast correction, while regularized off-policy updates learn elite self-generated algorithms, shifting the policy towards stronger neighbors.

We evaluate PCPO on the task of global placement (a critical procedure of the chip physical design process, deciding an overall layout of cells on the chip canvas (Markov et al., 2012)), designing learning rate schedules for a GPU-accelerated placer, DREAMPlace (Lin et al., 2020), which have been empirically shown to have a great impact on the final performance and convergence (Agnesina et al., 2023). Figure 1 shows the online performance comparisons, and PCPO achieves the largest improvement within acceptable computational cost in this online setting. Moreover, trained on only 4 chip cases, PCPO generalizes to other 12 chip cases, outperforming the state-of-the-art in-context evolutionary methods (e.g., OpenEvolve and ShinkaEvolve (Lange et al., 2026)) on average across 16 chip cases, and shows better sample efficiency than Group Relative Policy Optimization (GRPO) (Guo et al., 2025a). Based on an 8B-size model, PCPO achieves competitive performance compared to frontier closed-source models such as GPT-5.5, beating GPT-5.5 on training cases while slightly underperforming GPT-5.5 on testing cases. PCPO also reduces inference-time token cost by internalizing grounded domain knowledge and prompt distillation. After training, the policy can be decoded without thinking and self-generated references, thus saving tokens on a long chain of misleading rationales and prompts. We also evaluate PCPO on an extended application, GPU kernel design, in an online setting. PCPO achieves significant speedups on 4 cases of KernelBench (Ouyang et al., 2025), with an average 8.27× speedup against the PyTorch Eager baseline.

Our contributions can be summarized as three aspects:

(1) We empirically characterize in-context evolutionary stagnation in two specialized algorithm-design domains and present diagnostic evidence consistent with the hypothesis that repeatedly con-

ditioning on flawed self-generated context can reinforce unproductive generation patterns, reducing the probability of producing high-quality algorithms.

(2) We introduce the Improvement Chain as a motivating abstraction with a conditional local-transfer effect: under log-probability closeness and likelihood-gain assumptions, learning a visited algorithm increases a lower-bound change in the likelihood of a neighboring algorithm. Motivated by the conditional proposition, we propose PCPO, which has a curated global population to retain and reuse high-quality, diverse self-generated algorithms, refreshing on-policy updates to enhance exploration, and periodic regularized off-policy updates to shift the policy towards stronger neighbors.

(3) We evaluate PCPO on the task of learning rate schedule design for global placement and GPU kernel design. PCPO improves empirical performance and inference-time token efficiency over the state-of-the-art in-context LLM-driven evolutionary methods and is even competitive with frontier closed-source models, demonstrating the benefit of population-curated parametric updates.

2 Background

LLM-driven evolutionary search has become a standard recipe for algorithm design in open problems such as scientific discovery. Basically, in-context evolutionary methods have three key components: population, selection strategy, and operator, which function as external context management.

Population The population can be regarded as a delicate memory management mechanism that balances the quality and diversity of the produced algorithms: $\mathcal{A}_{\text{pop}} := \{(\boldsymbol{A}_i, \boldsymbol{I}_i)\}_{i=1}^{M}$, where $\boldsymbol{A}_i$ and $\boldsymbol{I}_i$ denote the i-th algorithm in the population and its corresponding information that includes execution feedback, respectively, and M is the population size.

Selection strategy The selection strategy can be regarded as a delicate selective context injection mechanism that selects the most important references and information from the memory needed by the agents at the current stage: $\{\boldsymbol{A}_{\text{ref}}^i, \boldsymbol{I}_{\text{ref}}^i\}_{i=1}^{k} = \text{select}(\mathcal{A}_{\text{pop}})$, where k denotes the number of selected algorithms.

Operator The operators C_{op} can be regarded as guiding instructions (e.g., explore new programs).

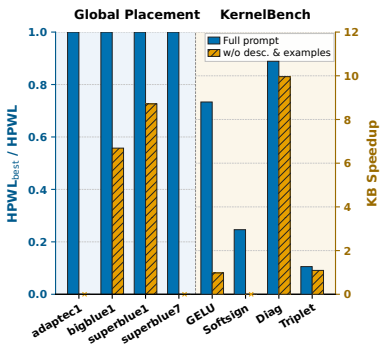
At iteration t, in-context evolutionary methods select reference algorithms $\{\boldsymbol{A}_{\text{ref}}^i, \boldsymbol{I}_{\text{ref}}^i\}_{i=1}^{k}$ and construct C_{op} to prompt the LLM to produce new algorithm designs that are later evaluated and written back into the population: $\boldsymbol{A}_{\text{new}} = \text{LLMs}(C_{\text{task}}, C_{\text{op}}, \{\boldsymbol{A}_{\text{ref}}^i, \boldsymbol{I}_{\text{ref}}^i\}_{i=1}^{k})$, where C_{task} is the context of the task prompt and $(C_{\text{op}}, \{\boldsymbol{A}_{\text{ref}}^i, \boldsymbol{I}_{\text{ref}}^i\}_{i=1}^{k})$ can be regarded as the additional context C_{extra}^t at step t.

3 Related Work

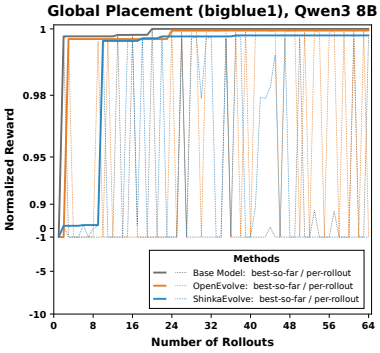
Evolutionary Algorithm Design: AlphaEvolve (Novikov et al., 2025) casts algorithm discovery as evolutionary search over an archive of programs, where a frozen LLM mutates selected parents and an island-based MAP-Elites archive balances quality against diversity. ShinkaEvolve (Lange et al., 2026) retains this loop and redesigns its parent selection, novelty rejection, and LLM routing. The evolutionary framework was first introduced by FunSearch (Romera-Paredes et al., 2024). EoH (Liu et al., 2024a) and ReEvo (Ye et al., 2024) extend it to heuristic code. ThetaEvolve (Wang et al., 2026) and TTT-Discover (Yuksekgonul et al., 2026) further internalize the search with test-time RL.

Off-Policy GRPO: On-policy GRPO discards each group after one update, which is costly when generation dominates the budget. Since a clipped off-policy GRPO surrogate still lower-bounds reward improvement when the behavior policy stays nearby (Mroueh et al., 2026), RePO (Li et al., 2025) adds an off-policy term to recover signal when on-policy advantages collapse. However, how the replay buffer is managed is first-order (Arnal et al., 2026): reuse is a trade-off among staleness and diversity, and a well-designed buffer can match on-policy accuracy at much lower cost.

Self-Evolving Agent: A self-evolving agent persistently rewrites its parameters, context, tools, or architecture from its own trajectories and feedback, aiming to improve future performance (Fang et al., 2025; Gao et al., 2026). Current systems do so mainly in three ways: evolving the harness (memory, tools, or workflows) (Zhang et al., 2026a); updating the model with SFT or RL on self-generated experience (Zhang et al., 2026c; Wang et al., 2026; Yuksekgonul et al., 2026); or selecting and mutating a population of programs at test-time (Chen et al., 2026).



(a) An illustration of relying on external domain knowledge rather than intrinsic knowledge.



(b) An illustration of the in-context evolutionary stagnation of OpenEvolve and ShinkaEvolve.

Figure 2: Diagnostic experiments based on Qwen3-8B.

4 From In-Context Stagnation to Parametric Self-Evolution

4.1 An Intuitive Analysis of In-Context Evolutionary Stagnation

Algorithms produced by LLMs in the form of canonicalized program representation can be modeled as $A_{\text{output}} = \arg\max_A P_{\text{LLM}}(A \mid C_{\text{task}})$ given the task context C_{task}, while in-context evolutionary methods inject operators, selected algorithms, and feedback as additional context C^t_{extra}, yielding $A_t = \arg\max_A P_{\text{LLM}}(A \mid C_{\text{task}}, C^t_{\text{extra}})$ at step t. Although this conditioning enables iterative search, it can sometimes stagnate when specialized domain knowledge is required to generate the optimal algorithm A^*. In particular,

$$\frac{P_{\text{LLM}}(A^* \mid C_{\text{task}}, C^t_{\text{extra}})}{P_{\text{LLM}}(A^* \mid C_{\text{task}})} = \frac{P_{\text{LLM}}(C^t_{\text{extra}} \mid C_{\text{task}}, A^*)}{P_{\text{LLM}}(C^t_{\text{extra}} \mid C_{\text{task}})}. \tag{1}$$

Because C^t_{extra} largely consists of the frozen model's previous outputs, misleading reasoning may be repeatedly reintroduced. Such context remains probable under the same model, but its probability may drop sharply conditioned on A^* if it is negatively associated with A^*, i.e., hallucinated self-generated content inside C^t_{extra} logically conflicts with the implementation of A^*. Under that condition, the probability of generating A^* using in-context evolutionary methods decreases compared to simple direct generations, creating a self-reinforcing cycle of biased outputs and local stagnation. Practical examples consistent with this hypothesis are provided in Appendix B.3.1.

We also construct some diagnostic experiments, whose results are consistent with the hypothesis. In Figure 2a, we compare the best results within 16 rollouts of the full prompt with the prompt that excludes detailed domain knowledge description, and "×" denotes failure to generate valid algorithms that satisfy the task constraints in all 16 rollouts. The performance drop relative to the full prompt shows that the base model relies on prompt-provided domain knowledge to improve performance rather than intrinsic knowledge on both tasks. Figure 2b illustrates the in-context evolutionary stagnation, where in-context evolutionary methods underperform the base model and fail to make improvements along the evolution process. One possible explanation for this performance drop is that self-generated context reinforces unproductive generation patterns. To show the generality of these phenomena, cross-model experiments are provided in Appendix D.

4.2 Improvement Chain as a Motivating Abstraction

As discussed in Section 4.1, repeatedly conditioning on misleading self-generated context may reduce the probability of producing high-quality algorithms in tasks that require domain knowledge. We therefore introduce the Improvement Chain as a conceptual model of parametric self-evolution, where, if each successor in the chain gradually approaches A^*, $P_{\text{LLM}}(\cdot \mid C_{\text{task}})$ is shifted towards more promising algorithms via parametric updates on curated self-generated algorithms, as shown in Figure 3. Then, the central question we face is that, if training increases the likelihood of a visited algorithm, under what conditions does this update also increase the likelihood of a nearby algorithm?

Setup: Let $\langle A_0, A_1, \ldots, A_n \rangle$ be an improvement chain with $A_n = A^*$. Note that in an ideal chain, A_{i+1} does not necessarily have a better fitness score than A_i, but is more promising with respect

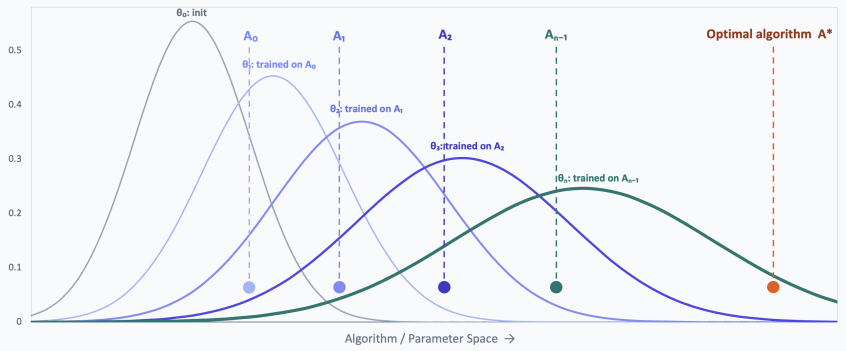


Figure 3: An illustration of the ideal Improvement Chain where each successor in the chain gradually approaches A^*. The optimal algorithm is approached by progressively increasing the reachability of successive chain nodes, generating and learning on chain nodes one after another.

to approaching the optimal A^*. This ordering represents a hypothesized target-directed relationship external to the proposition. For each chain step $i \in \{0, 1, \ldots, n-1\}$, let θ_i denote the model parameters before learning on A_i, and let the update operator (i.e., training θ_i with A_i) produce

$$\theta_{i+1} = \mathcal{U}(\theta_i; A_i). \tag{2}$$

For a fixed prompt context C, define

$$\ell_\theta(A \mid C) \triangleq \log p_\theta(A \mid C). \tag{3}$$

We assume that the chain nodes have positive probability under the policies considered, so that their log-probabilities are finite. Here, A denotes a fixed, canonicalized program representation.

From edit distance to probability closeness: We first formalize the algorithmic neighborhood in sequence space. Let $d(\cdot, \cdot)$ denote a token-level discrepancy metric over algorithms. We require that chain edges are short edits, which is common in real iterative refinement conducted by agents:

$$d(A_i, A_{i+1}) \le \Delta, \qquad \forall i \in \{0, 1, \ldots, n-1\}. \tag{4}$$

The probability of generating adjacent algorithms with no significant logical differences from the same model usually remains similar, motivating Assumption 1 for adjacent nodes in the chain.

Assumption 1 (Local log-probability closeness before and after the update)**.** *There exists $\kappa > 0$ such that, for every $i \in \{0, \ldots, n-1\}$ and $\theta \in \{\theta_i, \theta_{i+1}\}$,*

$$|\ell_\theta(A_i \mid C) - \ell_\theta(A_{i+1} \mid C)| \le \kappa \cdot d(A_i, A_{i+1}). \tag{5}$$

Since training the model on a visited algorithm increases the log-likelihood of generating it, we define ϵ_ℓ as the smallest of such gains along the chain. That is, for every $i \in \{0, \ldots, n-1\}$,

$$\ell_{\theta_{i+1}}(A_i \mid C) - \ell_{\theta_i}(A_i \mid C) \ge \epsilon_\ell. \tag{6}$$

Then, utilizing Eqs. 4 to 6, i.e., that the gap between the log-probabilities of A_i and A_{i+1} is upper bounded by $\kappa\Delta$ and an update raises the log-likelihood of A_i by at least ϵ_ℓ, we can derive Proposition 1, showing that the log-likelihood increment of generating A_{i+1} after an update is lower bounded by $\epsilon_\ell - 2\kappa\Delta$. Proofs are provided in Appendix A.

Proposition 1 (Local likelihood transfer)**.** *Under Assumptions 1, for every $i \in \{0, \ldots, n-1\}$,*

$$\ell_{\theta_{i+1}}(A_{i+1} \mid C) - \ell_{\theta_i}(A_{i+1} \mid C) \ge \epsilon_\ell - 2\kappa d(A_i, A_{i+1}) \ge \epsilon_\ell - 2\kappa\Delta. \tag{7}$$

Consequently, if $\rho_\ell \triangleq \epsilon_\ell - 2\kappa\Delta > 0$, then

$$p_{\theta_{i+1}}(A_{i+1} \mid C) \ge e^{\rho_\ell} p_{\theta_i}(A_{i+1} \mid C). \tag{8}$$

Therefore, Proposition 1 gives a sufficient condition under which training on the current algorithm increases the likelihood of a nearby algorithm. In particular, if the direct gain ϵ_ℓ on the log-likelihood of A_i exceeds twice the worst-case gap $\kappa\Delta$ between the log-probabilities of A_i and A_{i+1}, the successor receives a multiplicative probability improvement. If each successor in the chain gradually approaches A^*, such local probability transfer could increase the chance of sampling successors one after another, facilitating the traversal of the whole chain and shifting the task-conditional distribution of the base model towards the optimal algorithm. Appendix B.3.2 provides a practical example to show the actual magnitude of the values of the terms ϵ_ℓ and $\kappa\Delta$, satisfying $\rho_\ell > 0$.

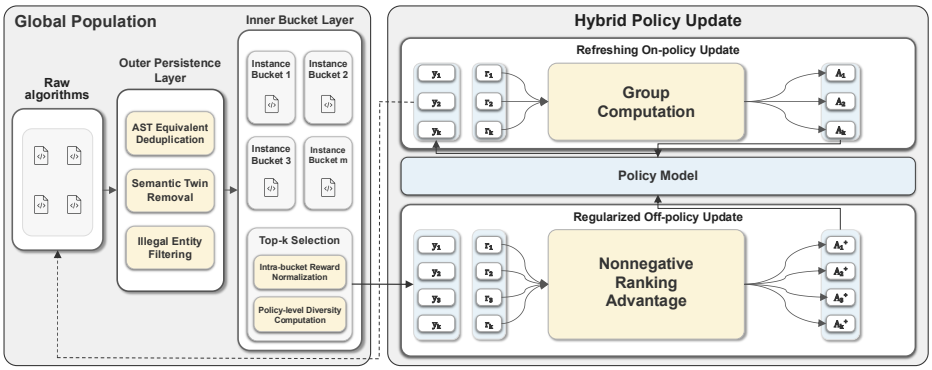


Figure 4: The workflow of PCPO. The left part illustrates the Global Population, filtering promising candidates while avoiding incompatible reward scales and difficulties. The right part illustrates the Hybrid Policy Update scheme, supporting sustained improvement, exploration, and rapid correction.

5 Practical Algorithm: Population-Curated Policy Optimization

From Ideal Chains to PCPO Section 4.2 provides a stylized account of one possible local-transfer effect but does not establish monotonic fitness improvement, guarantee the existence or discovery of an improvement chain, or provide a convergence guarantee. Instead, it motivates a practical design principle: retain useful algorithms, update the policy with promising elites, explore diverse variants in their neighborhoods, and shift the policy towards stronger algorithms. We therefore propose Population-Curated Policy Optimization as follows: retain a verified program A_i as a high-quality candidate in the global population, encourage exploration beyond near-duplicate programs by structural-diversity curation, approximate the likelihood-increasing update by positive off-policy reinforcement, and search for potential successors by fresh on-policy sampling.

5.1 Global Population

As shown in the left part of Figure 4, Global Population manages evaluated population experiences by retaining and reusing high-quality, diverse self-generated algorithms for focused learning. It consists of an Outer Persistence Layer and an Inner Bucket Layer.

The Outer Persistence Layer maintains a compact, quality-controlled archive $\mathcal{P}$, allowing valuable feedback to be reused beyond a single on-policy rollout. It removes redundant or invalid candidates through AST Equivalent Deduplication, Semantic Twin Removal, and Illegal Entity Filtering.

AST Equivalent Deduplication removes logically identical programs that differ only in presentation, such as comments, docstrings, or whitespace. For each code c, we parse its abstract syntax tree (AST), remove leading module/class/function docstrings, serialize the normalized tree, and hash it to obtain an equivalence key $\kappa(c)$. Codes sharing the same key are treated as duplicates, preventing superficial variants from consuming archive capacity or dominating Top-k selection.

Semantic Twin Removal prevents the archive from collapsing into near-identical elites after exact deduplication. We compute a structural distance $\mathrm{Div}_{\mathrm{struct}}(c, c')$ from ASTs normalized by docstring removal and α-renaming of bound identifiers. Given a keeper h (i.e., an elite already in the population) and a candidate g satisfying $\mathrm{Div}_{\mathrm{struct}}(g, h) \leq \delta_{\max}$, g is retained only if $\mathrm{fitness}(g) \geq \mathrm{fitness}(h) + \gamma_{\min}$. Thus, small structural variations are preserved only when they provide a meaningful fitness improvement, maintaining both archive quality and diversity.

Illegal Entity Filtering serves as a hard validity gate: programs with syntax or execution failures, or those violating task-specific constraints, are excluded from $\mathcal{P}$ and subsequent Top-k curation.

The Inner Bucket Layer partitions the global archive into instance-specific buckets $\{\mathcal{P}_b\}_{b=1}^{m}$, preventing incompatible reward scales and task difficulties from being mixed directly while retaining the sample-efficiency benefits of multi-instance training. Within each bucket, candidates are ranked for regularized off-policy learning using normalized reward and policy-level diversity.

Intra-bucket Reward Normalization rescales eligible fitness values to $[\varepsilon, 1]$ while preserving their within-bucket ranking. For a candidate $c \in \mathcal{P}_b$ with fitness r, let $\mathcal{R}_b$ denote the eligible rewards within the bucket b and define $r_{\min} = \min \mathcal{R}_b$ and $r_{\max} = \max \mathcal{R}_b$. Then we have

$$\tilde{r}_b(c) = \begin{cases} 1, & \text{if } r_{\max} = r_{\min}, \\ \varepsilon + (1-\varepsilon)\dfrac{r - r_{\min}}{r_{\max} - r_{\min}}, & \text{otherwise}, \end{cases} \tag{9}$$

where $\varepsilon > 0$ ensures a strictly positive quality score.

Policy-level Diversity Computation measures how structurally different an archived candidate is from codes recently generated by the current policy on the same instance (e.g., the same chip case):

$$d_b(c) = \begin{cases} \dfrac{1}{|\mathcal{C}_b^{(t)}|} \sum_{c' \in \mathcal{C}_b^{(t)}} \mathrm{Div}_{\mathrm{struct}}(c, c'), & \text{if } \mathcal{C}_b^{(t)} \neq \emptyset, \\ \mathrm{Div}_{\mathrm{struct}}(c,\, c_b^*), & \text{otherwise.} \end{cases} \tag{10}$$

where $\mathcal{C}_b^{(t)}$ is the set of distinct eligible codes generated by the current policy at step t for the instance bucket b, and c_b^* is the code with the best reward inside the bucket b. A larger $d_b(c) \in [0, 1]$ indicates greater novelty relative to the current policy.

Finally, candidates are ranked by the weighted score $s_b(c) = \alpha \cdot \tilde{r}_b(c) + \beta_{\text{div}} \cdot d_b(c)$, and the Top-$k$ candidates from each bucket are selected as elites for the regularized off-policy update.

5.2 Hybrid Policy Update

As shown in the right part of Figure 4, Hybrid Policy Update learns promising candidates curated by Global Population and shifts the policy towards stronger neighbors while exploring new ones. It combines a Refreshing On-policy Update, performed at every step for exploration, with a Regularized Off-policy Update, performed every T steps to reuse high-quality experiences.

Regularized Off-policy Update: Rather than using SFT (Ouyang et al., 2022; Zhang et al., 2026b) to imitate the token-level form of elite codes, we treat selected elites as off-policy trajectories and reinforce them through sequence-level advantages. This encourages the policy to learn transferable improvement patterns (Chu et al., 2025; Sun et al., 2026). Because stale off-policy samples can cause distribution drift and ratio explosion (Arnal et al., 2026; Mroueh et al., 2026), we retain clipped importance ratios and KL regularization toward a fixed reference policy. Selecting recent and diverse elites as candidates further limits sample staleness.

For the k elites in bucket b, sorted by decreasing environment return, we introduce a Nonnegative Ranking Advantage. Assigning rank weight $w_i = k - i + 1$ and setting $W = k(k+1)/2$, the advantage of elite i is $\hat{A}_i^+ = \hat{A}_0 \cdot \frac{w_i}{W} \cdot k = \hat{A}_0 \cdot \frac{2(k-i+1)}{k+1}$, $i = 1, \ldots, k$, $\hat{A}_0 = 1$. This positive rank profile has mean $\hat{A}_0$: stronger elites receive greater reinforcement, while every verified Top-k candidate retains a positive advantage, avoiding the negative feedback that standard within-group GRPO may assign to valid but lower-ranked elites. The off-policy objective is

$$\mathcal{J}_{\text{off}}(\theta) = \frac{1}{k} \sum_{i=1}^{k} \frac{1}{|c_i|} \sum_{t=1}^{|c_i|} \Big[\min\Big(r_{i,t}(\theta) A_i^+, \mathrm{clip}\big(r_{i,t}(\theta), 1-\varepsilon_{\text{clip}}, 1+\varepsilon_{\text{clip}}\big) A_i^+ \Big) - \beta D_{\mathrm{KL}}(\pi_\theta \| \pi_{\text{ref}}) \Big]. \tag{11}$$

where $r_{i,t}(\theta) = \frac{\pi_\theta(c_{i,t}|q, c_{i,<t})}{\pi_{\text{old},i}(c_{i,t}|q, c_{i,<t})}$, $\pi_{\text{old},i}$ generated elite c_i, and π_{ref} is the fixed reference policy.

Refreshing On-policy Update: At every step, Hybrid Policy Update applies standard GRPO with fresh rollouts and a smaller learning rate. Immediate environmental feedback promotes exploration and supplies new candidates to Global Population, while strong penalties rapidly suppress invalid or constraint-violating code. These fresh updates also anchor the policy to its current rollout distribution, mitigating the drift introduced by periodic off-policy updates.

6 Experiment

We evaluate our method on two algorithm-design scenarios, i.e., global placement in EDA (Chu, 2009) and the extended experiment, GPU kernel design (Ouyang et al., 2025). We adopt Qwen3-8B (Yang et al., 2025) as the base model and equip it with in-context evolutionary harnesses like OpenEvolve (the open-source implementation of AlphaEvolve (Novikov et al., 2025)) and ShinkaE-

Table 1: Best@n HPWL ($\times 10^6$) results on two global placement benchmarks. The lower, the better. The 4 cases used for RL training are underlined.

	Qwen3-8B			OpenEvolve			ShinkaEvolve			GRPO			PCPO		
	best@1	best@4	best@16	best@1	best@4	best@16	best@1	best@4	best@16	best@1	best@4	best@16	best@1	best@4	best@16
adaptec1	71.11	71.09	71.01	×	×	70.54	×	71.58	71.58	×	71.22	70.51	**70.80**	**70.36**	**70.15**
adaptec2	×	×	×	×	80.27	79.35	82.64	81.04	81.04	80.18	80.11	80.11	**80.12**	**78.83**	**78.83**
adaptec3	×	×	187.64	234.01	233.98	233.98	**187.71**	187.71	187.71	×	187.90	186.94	187.82	**186.40**	**186.25**
adaptec4	171.45	171.45	171.36	206.43	206.43	206.43	175.33	173.18	173.18	172.12	171.69	170.55	**171.40**	**171.40**	**168.26**
bigblue1	**88.45**	88.22	88.20	×	88.90	88.90	×	208.83	89.21	88.75	88.64	87.67	88.59	**88.06**	**87.65**
bigblue2	×	133.03	132.94	**133.04**	133.00	132.97	×	133.00	132.41	×	133.07	132.45	×	**132.93**	**132.28**
bigblue3	×	326.46	326.39	×	×	×	345.57	**303.47**	303.47	×	326.42	323.87	**328.14**	326.58	**303.11**
bigblue4	×	×	×	757.67	757.61	757.53	**732.25**	**732.25**	732.25	×	756.62	756.62	755.41	748.17	**727.46**
superblue1	×	**388.12**	387.80	**389.19**	389.19	389.19	×	468.06	468.06	536.79	388.94	388.12	389.67	388.95	**385.68**
superblue3	×	469.40	469.40	**462.31**	462.31	460.57	×	×	×	465.64	461.37	461.23	463.14	**461.36**	**455.29**
superblue4	×	291.76	291.76	×	×	×	×	×	×	×	296.83	290.60	**291.58**	**291.58**	**289.79**
superblue5	×	454.36	452.02	×	459.10	459.10	×	×	×	**452.33**	452.33	451.94	457.61	**452.20**	**450.76**
superblue7	728.17	723.62	723.62	556.00	556.00	556.00	×	×	×	707.16	707.16	549.55	**551.07**	**549.45**	**547.43**
superblue10	862.16	862.08	862.02	861.26	861.26	861.25	×	863.92	863.92	861.51	861.51	859.15	**860.22**	**860.22**	**856.73**
superblue16	401.68	401.36	401.32	401.35	401.35	401.35	×	×	×	**401.03**	401.03	400.68	401.59	**400.73**	**400.31**
superblue18	233.75	233.35	233.11	223.88	223.88	223.84	×	×	224.45	232.18	223.78	223.14	**223.47**	**223.47**	**222.82**
Avg rank	3.66	3.38	3.69	2.91	3.62	3.78	3.72	3.94	4.16	3.06	2.69	2.38	**1.66**	**1.38**	**1.00**

volve (Lange et al., 2026). We also provide GRPO (Guo et al., 2025a) as a comparable baseline enhanced by reinforcement learning. Moreover, we further compare our method to frontier closed-source models such as GPT-5.5, as well as a general-purpose coding agent harness, Codex. Note that since ThetaEvolve (Wang et al., 2026) is a test-time learning method, we provide global placement results in an online setting in Appendix B.3.3. Cross-model results are shown in Appendix D.

6.1 Global Placement

Global placement (GP) is a crucial step in chip placement (Chu, 2009), which has a direct and significant impact on the performance, power consumption, and area (PPA) of the final chip design, as it determines the locations of the cells within the given chip layout area (Markov et al., 2012). The goal of GP is to minimize half-perimeter wirelength (HPWL) while meeting the target overflow. In this experiment, we apply algorithm-design agents to design the learning rate schedules for the Adam optimizer (Kingma & Ba, 2014) of the GPU-accelerated placer DREAMPlace (Lin et al., 2020), which, as shown empirically, has a great impact on the quality and convergence of the GP process and is hard to tune (Agnesina et al., 2023). Designing the learning rate schedules requires domain knowledge, balancing the wire-length and cell density via changing the positions of cells through gradient backpropagation (Lin et al., 2020). More detailed descriptions of GP are in Appendix B.1.

As shown in Table 1, we test PCPO on two popular open-source benchmarks for GP, ISPD 2005 (Nam et al., 2005) and ICCAD 2015 (Kim et al., 2015). The metric best@n shows the best result achieved within n rollouts. "×" denotes that no valid result satisfying the task constraints is achieved within n rollouts. For RL methods, models are trained on 4 representative chip cases within their own series and are then directly used for inference on all cases. PCPO achieves the best average rank on all best@n metrics, which demonstrates its improvement within a small rollout budget and its stability on one-shot generations. These results indicate that PCPO effectively retains and reuses valuable and diverse experiences for parametric learning, inducing generalizable capabilities. Note that in-context evolutionary methods show worse average performance compared to the base model, which is consistent with the hypothesis made in Section 4.1. Figure 10 further shows PCPO's sample efficiency in mixed-case training brought by population experience reuse compared to GRPO. PCPO also outperforms RePO (Li et al., 2025), a hybrid-update RL method, suggesting the importance of balancing the quality and diversity of population experiences, as shown in Table 8. As shown in Figure 11, PCPO also reduces inference-time token cost by internalizing grounded domain knowledge (no need to perform long-range reasoning) and distilling the detailed prompt (e.g., in-context exemplars that evolutionary methods repeatedly inject). Thus, the cost of training is worthwhile because after a single training run, PCPO can achieve outstanding performance on broader cases in the specific domain with minimal token consumption. More ablation results are in Appendix B.3.4.

Moreover, to demonstrate the effectiveness of self-evolving via PCPO, we compare PCPO to baseline results (DREAMPlace (Lin et al., 2020) with human-set parameters), as well as GPT-5.5, GPT-4o, and Codex. We use the average best@16 of 16 independent runs for a fair and stable comparison. As shown in Table 2, trained on only 4 cases, PCPO outperforms baseline results with an average improvement of 1.54%, surpassing GPT-4o and Codex. Compared to GPT-5.5, PCPO also achieves

Table 2: Average best@16 HPWL ($\times 10^6$) of 16 independent runs and relative improvement. The 4 cases used for RL training are underlined.

Chip case	DREAMPlace	GPT-4o		GPT-5.5		Codex-GPT-5.5		PCPO-Qwen3-8B	
	HPWL	best@16	Imp. (%)	best@16	Imp. (%)	best@16	Imp. (%)	best@16	Imp. (%)
adaptec1	73.27	71.04	+3.04%	70.32	+4.03%	73.61	-0.46%	**70.18**	**+4.22%**
adaptec2	82.28	84.63	-2.86%	80.16	+2.58%	79.84	+2.96%	**78.92**	**+4.08%**
adaptec3	189.17	186.20	+1.57%	186.19	+1.57%	203.45	-7.55%	**186.14**	**+1.60%**
adaptec4	172.15	**167.56**	**+2.67%**	169.12	+1.76%	230.42	-33.85%	168.34	+2.21%
bigblue1	88.93	**87.28**	**+1.86%**	87.54	+1.56%	169.12	-90.18%	87.43	+1.69%
bigblue2	133.26	**131.91**	**+1.01%**	**131.91**	**+1.01%**	132.54	+0.54%	132.24	+0.77%
bigblue3	305.24	340.30	-11.49%	**292.92**	**+4.04%**	296.77	+2.78%	302.98	+0.74%
bigblue4	737.65	764.68	-3.66%	736.09	+0.21%	837.54	-13.54%	**727.29**	**+1.40%**
superblue1	390.08	389.08	+0.26%	385.69	+1.12%	390.71	-0.16%	**385.56**	**+1.16%**
superblue3	462.22	445.43	+3.63%	**444.38**	**+3.86%**	512.66	-10.91%	455.34	+1.49%
superblue4	292.11	289.07	+1.04%	289.25	+0.98%	**288.97**	**+1.07%**	289.82	+0.78%
superblue5	452.34	451.06	+0.28%	**449.88**	**+0.54%**	451.11	+0.27%	450.57	+0.39%
superblue7	556.69	547.54	+1.64%	551.60	+0.91%	549.66	+1.26%	**547.21**	**+1.70%**
superblue10	861.29	861.12	+0.02%	856.78	+0.52%	856.82	+0.52%	**856.52**	**+0.55%**
superblue16	402.58	400.54	+0.51%	**399.76**	**+0.70%**	399.97	+0.65%	400.08	+0.62%
superblue18	224.86	230.06	-2.31%	222.82	+0.91%	**219.07**	**+2.58%**	221.94	+1.30%
Avg-train	1.000	0.983	+1.70%	0.981	+1.91%	1.224	-22.38%	**0.978**	**+2.19%**
Avg-test	1.000	1.008	-0.80%	**0.984**	**+1.56%**	1.045	-4.54%	0.987	+1.33%
Avg	1.000	1.002	-0.17%	**0.984**	**+1.64%**	1.090	-9.00%	0.985	+1.54%

Table 3: Average best speedups of 4 independent runs on 4 cases of KernelBench. Higher is better.

Method	MinGPTNewGelu	SoftSign	MatmulWithDiagonalMatrices	TripletMarginLoss	Avg rank
Qwen3-8B	8.801×	2.957×	10.664×	1.267×	6.00
OpenEvolve	8.741×	3.373×	10.478×	1.277×	5.25
ShinkaEvolve	8.727×	3.373×	10.478×	1.094×	6.00
ThetaEvolve	8.986×	3.306×	15.960×	4.524×	4.00
GPT-5.5	9.018×	2.939×	**16.621**×	4.537×	3.38
Codex	8.712×	3.350×	**16.621**×	4.419×	4.38
GRPO	9.002×	3.071×	10.169×	3.400×	5.50
PCPO	**9.033**×	**3.446**×	16.067×	**4.551**×	**1.50**

comparable average results (i.e., beating GPT-5.5 on training cases while slightly underperforming it on testing cases), with better results on 10 cases based on the 8B base model. Note that Codex can be seen as another in-context evolutionary method because it automatically selects candidate codes and improves them. The performance degradation is also consistent with the hypothesis that hallucinated self-generated context can reinforce a cycle of biased outputs and local stagnation.

6.2 Extended Experiment: GPU Kernel Design

GPU kernel design is a hardware-aware optimization process that maps high-level tensor computations onto the parallel execution and memory hierarchy of a target accelerator, greatly influencing the GPU computational efficiency and energy consumption (Tillet et al., 2019; Spector et al., 2025). We apply algorithm-design agents to design GPU kernels for 4 Level-1 cases highlighted in the original KernelBench paper (Ouyang et al., 2025), covering distinct computational structures: transcendental-heavy elementwise computation, bandwidth-dominated pointwise computation, structured broadcasting, and multi-input reduction. We use the best speedup within the rollout budget as the metric. For RL methods, we test their inference-time learning ability by granting them the same rollout budget as in-context methods during online training. More detailed descriptions and experiment settings can be found in Appendix C. As demonstrated in Table 3, PCPO achieves the best average rank, which shows the effectiveness of parametric self-evolution at inference time.

7 Conclusion

In this paper, we develop self-evolving algorithm-design agents that evolve via exploring and training on self-generated algorithms with population experiences. Our proposed PCPO improves results across two algorithm-design tasks (i.e., GP in EDA and GPU kernel design) over state-of-the-art in-context evolutionary methods, remains competitive with frontier closed-source models based on Qwen3-8B, and reduces inference-time token cost, showing the benefit of population-curated parametric self-evolution for algorithm-design agents rather than stagnating with in-context evolution.

AI USE STATEMENT

(This section is **required** and does not count toward the page limit.)

In this work, we used generative AI tools to assist with translation. We have not used generative AI tools to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, propose or refine hypotheses, design or provide feedback on research methodology or experiments, implement methods, clean and reformat datasets, support qualitative and thematic data analysis, or interpret results. We have reviewed all AI-assisted work, i.e., we checked the translations and made sure they conform to the original meaning. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

REPRODUCIBILITY STATEMENT

Our code is available at `https://anonymous.4open.science/r/PCPO-222D/`. All the proofs are provided in Appendix A. All datasets used in the paper can be found through the links provided in their original papers.

REFERENCES

Anthony Agnesina, Puranjay Rajvanshi, Tian Yang, Geraldo Pradipta, Austin Jiao, Ben Keller, Brucek Khailany, and Haoxing Ren. Autodmp: Automated dreamplace-based macro placement. In *Proceedings of the 2023 International Symposium on Physical Design*, Virtual Event, 2023.

Charles Arnal, Vivien Cabannes, Taco Cohen, Julia Kempe, and Remi Munos. Efficient RL training for LLMs with experience replay. *arXiv:2604.08706*, 2026.

Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Ben Mann, Nova DasSarma, et al. A general language assistant as a laboratory for alignment. *arXiv:2112.00861*, 2021.

Alan Blount, Antonio Gulli, Shubham Saboo, Michael Zimmermann, and Vladimir Vuskovic. Introduction to agents. Whitepaper, Google, 2025. URL `https://www.kaggle.com/whitepaper-introduction-to-agents`.

Yao-Wen Chang, Zhe-Wei Jiang, and Tung-Chieh Chen. Essential issues in analytical placement algorithms. *IPSJ Transactions on System LSI Design Methodology*, 2:145–166, 2009.

Jiacheng Chen, Xinyu Zhang, Shunkai Zhang, Yanmohan Wang, Lin Li, Tiancheng Qin, Qin Wang, Zhengmao Zhu, Tianle Li, Jingyang Li, et al. Maxproof: Scaling mathematical proof with generative-verifier RL and population-level test-time scaling. *arXiv:2606.13473*, 2026.

Tung-Chieh Chen, Zhe-Wei Jiang, Tien-Chang Hsu, Hsin-Chen Chen, and Yao-Wen Chang. NTUplace3: An analytical placer for large-scale mixed-size designs with preplaced blocks and density constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1228–1240, 2008.

Chung-Kuan Cheng, Andrew B Kahng, Ilgweon Kang, and Lutong Wang. Replace: Advancing solution quality and routability validation in global placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(9):1717–1730, 2018.

Chris Chu. Placement. In *Electronic Design Automation*, pp. 635–685. Elsevier, 2009.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. SFT memorizes, RL generalizes: A comparative study of foundation model post-training. In *Proceedings of the 42nd International Conference on Machine Learning*, Vancouver, Canada, 2025.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, et al. A comprehensive survey of self-evolving AI agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv:2508.07407*, 2025.

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: What, when, how, and where to evolve on the path to artificial super intelligence. *Transactions on Machine Learning Research*, 2026-January, 2026.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. Deepseek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv:2501.12948*, 2025a.

Yiduo Guo, Jie Fu, Huishuai Zhang, and Dongyan Zhao. Efficient domain continual pretraining by mitigating the stability gap. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, Vienna, Austria, 2025b.

Zixuan Ke, Yifei Ming, Xuan-Phi Nguyen, Caiming Xiong, and Shafiq Joty. Demystifying domain-adaptive post-training for financial LLMs. In *Proceedings of the 30th Conference on Empirical Methods in Natural Language Processing*, Suzhou, China, 2025.

Myung-Chul Kim, Jin Hu, Jiajia Li, and Natarajan Viswanathan. ICCAD-2015 CAD contest in incremental timing-driven placement and benchmark suite. In *Proceedings of the 2015 International Conference on Computer-Aided Design*, Austin, TX, 2015.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv:1412.6980*, 2014.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Siheng Li, Zhanhui Zhou, Wai Lam, Chao Yang, and Chaochao Lu. Repo: Replay-enhanced policy optimization. *arXiv:2506.09340*, 2025.

Xiaoya Li, Xiaofei Sun, Albert Wang, Jiwei Li, and Chris Shum. Cuda-l1: Improving CUDA optimization via contrastive reinforcement learning. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026a.

Yibo Li, Zijie Lin, Ailin Deng, Xuan Zhang, Yufei He, Shuo Ji, Tri Cao, and Bryan Hooi. Just-in-time reinforcement learning: Continual learning in LLM agents without gradient updates. In *Proceedings of the 43rd International Conference on Machine Learning*, Seoul, South Korea, 2026b.

Yibo Lin, Zixuan Jiang, Jiaqi Gu, Wuxi Li, Shounak Dhar, Haoxing Ren, Brucek Khailany, and David Z Pan. DREAMPlace: Deep learning toolkit-enabled gpu acceleration for modern VLSI placement. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40 (4):748–761, 2020.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, Vienna, Austria, 2024a.

Gang Liu, Yihan Zhu, Jie Chen, and Meng Jiang. Scientific algorithm discovery by augmenting alphaevolve with deep research. *arXiv:2510.06056*, 2025a.

Kai Liu, Ze Chen, Zhihang Fu, Wei Zhang, Rongxin Jiang, Fan Zhou, Yaowu Chen, Yue Wu, and Jieping Ye. Structure-aware domain knowledge injection for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, Vienna, Austria, 2025b.

Lixin Liu, Bangqi Fu, Shiju Lin, Jinwei Liu, Evangeline F. Y. Young, and Martin D. F. Wong. Xplace: An extremely fast and extensible placement framework. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(6):1872–1885, 2024b.

Chen Lu, Ke Xue, Chengrui Gao, Yunqi Shi, Siyuan Xu, Mingxuan Yuan, Chao Qian, and Zhi-Hua Zhou. Advancing automated algorithm design via evolutionary stagewise design with llms. *arXiv:2603.07970*, 2026.

Jingwei Lu, Pengwen Chen, Chin-Chih Chang, Lu Sha, Dennis Jen-Hsin Huang, Chin-Chi Teng, and Chung-Kuan Cheng. ePlace: Electrostatics-based placement using fast Fourier transform and Nesterov's method. *ACM Transactions on Design Automation of Electronic Systems*, 20(2):1–34, 2015.

Igor L Markov, Jin Hu, and Myung-Chul Kim. Progress and challenges in VLSI placement research. In *Proceedings of the 25th International Conference on Computer-Aided Design*, San Jose, CA, 2012.

Youssef Mroueh, Nicolas Dupuis, Brian Belgodere, Apoorva Nitsure, Mattia Rigotti, Kristjan Greenewald, Jiri Navratil, Jarret Ross, and Jesus Rios. Revisiting group relative policy optimization: Insights into on-policy and off-policy training. In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Gi-Joon Nam, Charles J Alpert, Paul Villarrubia, Bruce Winter, and Mehmet Yildiz. The ISPD2005 placement contest and benchmark suite. In *Proceedings of the 2005 International Symposium on Physical Design*, San Francisco, CA, 2005.

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:2506.13131*, 2025.

Anne Ouyang, Simon Guo, Simran Arora, Alex L Zhang, William Hu, Christopher Re, and Azalia Mirhoseini. Kernelbench: Can LLMs write efficient GPU kernels? In *Proceedings of the 42nd International Conference on Machine Learning*, Vancouver, Canada, 2025.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 36*, New Orleans, LA, 2022.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context. *arXiv:2209.15189*, 2022.

Benjamin Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Dan Fu, and Christopher Re. Thunderkittens: Simple, fast, and adorable kernels. In *Proceedings of the 13th International Conference on Learning Representations*, Singapore, 2025.

Yiyou Sun, Yuhan Cao, Pohao Huang, Haoyue Bai, Hanna Hajishirzi, Nouha Dziri, and Dawn Song. RL grokking recipe: How does RL unlock and transfer new algorithms in LLMs? In *Proceedings of the 14th International Conference on Learning Representations*, Rio de Janeiro, Brazil, 2026.

Gemma Team, Sherif El Abd, Vaibhav Aggarwal, Robin Algayres, Alek Andreev, Olivier Bachem, Ian Ballantyne, Cormac Brick, Victor Cărbune, Michelle Casbon, et al. Gemma 4 technical report. *arXiv:2607.02770*, 2026.

Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, Phoenix, AZ, 2019.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. Thetaevolve: Test-time learning on open problems. In *Proceedings of the 43rd International Conference on Machine Learning*, Seoul, South Korea, 2026.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents: A survey. *Science China information sciences*, 68(2):121101, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv:2505.09388*, 2025.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. ReEvo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems 38*, Vancouver, Canada, 2024.

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:2601.16175*, 2026.

Guibin Zhang, Haotian Ren, Chong Zhan, Zhenhong Zhou, Junhao Wang, He Zhu, Wangchunshu Zhou, and Shuicheng Yan. Memevolve: Meta-evolution of agent memory systems. In *Proceedings of the 43rd International Conference on Machine Learning*, Seoul, South Korea, 2026a.

Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Guoyin Wang, et al. Instruction tuning for large language models: A survey. *ACM Computing Surveys*, 58(7):1–36, 2026b.

Yudi Zhang, Meng Fang, Zhenfang Chen, and Mykola Pechenizkiy. Self-evolving LLM agents with in-distribution optimization. In *Proceedings of the 43rd International Conference on Machine Learning*, Seoul, South Korea, 2026c.

Yunxiang Zhang, Ping Yu, Jianyu Wang, Max (Xiangjun) Fan, Julian Reed, Azalia Mirhoseini, Will Su, et al. Kernelbench-verified: Do LLM-generated kernels actually beat PyTorch? *arXiv:2607.16241*, 2026d.

Table 4: Hyperparameters of PCPO with Qwen3-8B on Global Placement. Every T on-policy steps, we additionally run an off-policy update on the global-population top-k programs ranked by composite score $S = \alpha\,\tilde{r} + \beta\,d$, where $\tilde{r}$ is min–max normalized reward and d is semantic diversity. Near-duplicates with diversity $\leq \delta_{\max}$ are merged unless the fitness gain is at least $\gamma_{\min}$.

Setting	**Symbol / name**	**Value**
Reward and legality		
Illegal / crash / missing API	—	-10^9
Not achieving target overflow	—	-1000
Valid placement	r	DREAMPlace `get_fitness` (higher better)
Illegal threshold (crash band)	r_{inv}	-10^8
PCPO		
Off-policy interval	T	2
Population top-k	k	4
Composite reward weight	α	1.0
Composite diversity weight	β	1.0
Near-duplicate diversity cutoff	$\delta_{\max}$	0.15
Min. fitness gain to replace near-dup.	$\gamma_{\min}$	0.01
Replay buffer size	—	512
On-policy learning rate	lr_{on}	1×10^{-6}
Off-policy learning rate	lr_{off}	2×10^{-6}
Optimization		
Train batch size (prompts)	B	4
Group size (GRPO)	n	4
PPO epochs	—	1
KL coefficient (low-var. KL)	β_{KL}	0.001
Clip ratio / entropy / grad. clip	—	0.2 / 0.001 / 1.0
Training epochs / steps	—	1 / 16
Dataset size	—	64 prompts
Max prompt / response tokens	—	2048 / 2048
Decoding		
Temperature / top-p	—	0.5 / 1.0
Max new tokens	—	1024
Qwen3 thinking mode	—	off

A PROOF

Proposition 1 (Local likelihood transfer)**.** *Under Assumptions 1, for every* $i \in \{0, \ldots, n-1\}$*,*

$$\ell_{\theta_{i+1}}(A_{i+1} \mid C) - \ell_{\theta_i}(A_{i+1} \mid C) \geq \epsilon_\ell - 2\kappa d(A_i, A_{i+1}) \geq \epsilon_\ell - 2\kappa\Delta. \tag{12}$$

Consequently, if $\rho_\ell \triangleq \epsilon_\ell - 2\kappa\Delta > 0$*, then*

$$p_{\theta_{i+1}}(A_{i+1} \mid C) \geq e^{\rho_\ell} p_{\theta_i}(A_{i+1} \mid C). \tag{13}$$

Proof. Let $\theta = \theta_i$ and $\theta' = \theta_{i+1}$. By Assumption 1,

$$\begin{aligned}
&\ell_{\theta'}(A_{i+1} \mid C) - \ell_\theta(A_{i+1} \mid C) \\
&= \big[\ell_{\theta'}(A_{i+1} \mid C) - \ell_{\theta'}(A_i \mid C)\big] + \big[\ell_{\theta'}(A_i \mid C) - \ell_\theta(A_i \mid C)\big] \\
&\quad + \big[\ell_\theta(A_i \mid C) - \ell_\theta(A_{i+1} \mid C)\big] \\
&\geq -\kappa d(A_i, A_{i+1}) + \epsilon_\ell - \kappa d(A_i, A_{i+1}),
\end{aligned} \tag{14}$$

which proves Eq. 7. Exponentiating both sides gives Eq. 8. □

B GLOBAL PLACEMENT

B.1 INTRODUCTION TO GLOBAL PLACEMENT

Global placement (GP) is a critical step of the overall chip placement procedure (Chu, 2009), which has a direct and significant impact on the performance, power consumption, and area (PPA) of the

final chip design, as it determines the locations of the cells within the given chip layout area (Markov et al., 2012). GP aims to deliver a high-quality rough overall layout for all cells and is required to meet the target overflow, a metric that measures a chip's total overlap. It should also achieve the lowest possible half-perimeter wirelength (HPWL) and run as fast as possible. (Lin et al., 2020). For formulation, GP is typically defined as a constrained optimization problem. It uses HPWL as the objective and density as the constraint.:

$$\begin{aligned} \min_{\mathbf{x},\mathbf{y}} \text{HPWL}(\mathbf{x},\mathbf{y}) = \min_{\mathbf{x},\mathbf{y}} \sum\nolimits_{e\in E} \text{HPWL}_e(\mathbf{x},\mathbf{y}), \\ \text{s.t. } \text{D}(\mathbf{x},\mathbf{y}) \leq d_t, \end{aligned} \tag{15}$$

where $(\mathbf{x},\mathbf{y})$ denotes the 2-D positions of all the cells, E denotes the set of nets, $\text{HPWL}_e(\mathbf{x},\mathbf{y}) = (\max_{i\in e}x_i - \min_{i\in e}x_i) + (\max_{i\in e}y_i - \min_{i\in e}y_i)$ denotes the HPWL of each net e, $\text{D}(\mathbf{x},\mathbf{y})$ denotes the density of each location of the layout, and d_t denotes the target density. Note that a net covers multiple cells and represents how these cells connect with one another during the routing phase.

Existing GP methods: Analytical placement (Chang et al., 2009) represents one class of state-of-the-art GP methods. Among these approaches, nonlinear placement (Chen et al., 2008; Lu et al., 2015; Cheng et al., 2018) delivers the best performance. Nonlinear placement generally addresses the constrained GP optimization problem defined in Eq. (15) using the penalty method. Specifically, it solves a series of Lagrangian-relaxation problems with progressively increasing Lagrangian multipliers, as formulated below:

$$\min_{\mathbf{x},\mathbf{y}} \left(\sum\nolimits_{e\in E} \text{WL}_e(\mathbf{x},\mathbf{y})\right) + \lambda \cdot \text{D}(\mathbf{x},\mathbf{y}), \tag{16}$$

where $\text{WL}_e(\mathbf{x},\mathbf{y})$ is a smooth approximation of $\text{HPWL}_e(\mathbf{x},\mathbf{y})$, and λ is the Lagrangian multiplier (also known as density weight), which controls the weight of the density penalty function $\text{D}(\mathbf{x},\mathbf{y})$. To achieve smoother optimization, ePlace (Lu et al., 2015) and RePlace (Cheng et al., 2018) calculate a density penalty function analogous to the potential energy in an electrostatic system. In this model, cells act as charges, and the density gradient represents the electric field. This penalty function is smoother than the original density function and maintains strong correlation with it.

Despite their outstanding performance, nonlinear methods are typically time-consuming. For this reason, they cannot satisfy the fast-verification requirements of industrial scenarios, particularly for large-scale chip cases (Lin et al., 2020). To address this issue, acceleration algorithms such as DREAMPlace (Lin et al., 2020) and Xplace (Liu et al., 2024b) have been developed. These algorithms leverage modern GPUs and deep-learning toolkits to speed up the gradient-descent optimization process. Specifically, DREAMPlace (Lin et al., 2020) incorporates modern gradient-descent optimizers (e.g., Adam, Nesterov, etc.) and adapts neural-network training techniques to placement tasks. It delivers strong performance and stands as one of the most widely used and advanced open-source placers available today. However, as shown empirically, the learning rate schedules have a huge impact on the quality and convergence of the GP process and are hard to tune (Agnesina et al., 2023). Designing the learning rate schedules requires OOD domain knowledge since they need to balance the wire-length and cell density via changing the positions of cells through gradient backpropagation (Lin et al., 2020). For the ISPD 2005 benchmark, the target overflow is 0.07; for the ICCAD 2015 benchmark, the target overflow is 0.1.

B.2 Hyperparameters

We provide the training hyperparameters of PCPO in Table 4. All RL methods have the same rollout budget in the training process.

B.3 Extended results

B.3.1 Practical examples for diagnostic experiments of in-context evolutionary stagnation

On bigblue1, we inspect OpenEvolve's thinking traces with the base model, Qwen3-8B. A representative excerpt is shown in Fig. 5a. The model does not merely mistune constants; it searches a qualitatively inverted learning-rate policy relative to a strong direct sample from the same model family (Fig. 5c, fitness -87.26, overflow 0.068).

Qwen3-8B chain-of-thought (OpenEvolve on `bigblue1`, parent fitness -1000)

The current program is using a baseline exponential decay . . . But the fitness is still -1000, which means the overflow is above 0.07.
*Looking at the previous attempts, **[H1]** the problem seems to be that the learning rate isn't decreasing enough when overflow is high. The current code reduces the learning rate by 0.99 when overflow is above 0.07, but maybe that's not enough. . . . multiplying by 0.95 instead of 0.99 when overflow is too high. That would make the learning rate drop faster, which might help in reducing overflow more quickly.*

*Wait, the `log_hpwl_prev` and `log_hpwl` could indicate if the hpwl is improving. **[H2]** If `log_hpwl` is decreasing (meaning hpwl is getting worse), maybe we need to adjust the learning rate. But in the current code, if `log_hpwl` is less than `log_hpwl_prev`, it's considered an improvement, so it increases the learning rate.*

***[H3]** if `log_gradient_norm` > `log_gradient_norm_prev`: reduce the learning rate. Wait, but `log_gradient_norm_prev` is not a parameter. . . . So, perhaps using the current `log_gradient_norm` as a signal. For example, if `log_gradient_norm` is higher than some threshold, reduce the learning rate.*

H1: inverted causality (overflow stays high ⇒ decay even faster, compounding on `learning_rate_prev`). H2: inverted semantics (↓ log HPWL means HPWL *improves*, not worsens). H3: phantom variable (the API has no previous gradient norm).

(b) OpenEvolve child of (a)

fitness -1000; final overflow 0.86 (illegal)

```
def adjust_learning_rate(
    init_learning_rate, step_num,
     log_hpwl,
    log_hpwl_prev, overflow, log_lambda,
    learning_rate_prev,
     log_gradient_norm):
    lr = learning_rate_prev * 0.995
    if overflow > 0.07:
        lr *= 0.95          # extra
     compounding decay
    elif log_hpwl < log_hpwl_prev:
        lr *= 1.002         # HPWL down
     => raise LR
    elif log_gradient_norm > 5.0:
        lr *= 0.99          # invented
     threshold
    else:
        lr *= 0.995
    return lr
```

(c) Direct sample (same task; best of 64)

fitness -87.26; final overflow 0.068 (legal)

```
def adjust_learning_rate(
    init_learning_rate, step_num, log_hpwl,
    log_hpwl_prev, overflow, log_lambda,
    learning_rate_prev, log_gradient_norm):
    decay = 0.995 ** step_num           #
     scheduled, not recursive
    hpwl_f = 1.0 + 0.5 * (log_hpwl -
     log_hpwl_prev)
    ov_f   = 1.0 - 0.5 * max(overflow -
     0.07, 0.0)
    g_f    = 1.0 / (1.0 + 0.5 *
     np.exp(log_gradient_norm))
    lam_f  = 1.0 / (1.0 + 0.2 *
     np.exp(log_lambda))
    return (init_learning_rate * decay
            * hpwl_f * ov_f * g_f * lam_f)
```

Figure 5: A representative OpenEvolve thinking trace on `bigblue1` versus a strong direct generation based on Qwen3-8B. Panel (a) is lightly abridged from the model's `<think>` block; highlighted claims H1–H3 are factually false or internally inconsistent. Panel (b) is the program emitted from that trace: extra multiplicative decay on `learning_rate_prev` while overflow exceeds 0.07 compounds into $\eta_t = \eta_0(0.995 \cdot 0.95)^t$, the optimizer stalls, and overflow never reaches the 0.07 target. Panel (c) reconstructs the expert schedule $\eta_0 \cdot 0.995^t$ and applies non-compounding state-dependent modulators; HPWL improving (`log_hpwl` < `log_hpwl_prev`) *shrinks* the step, the opposite of (b). Ellipses in (a) mark omitted sentences; wording is otherwise verbatim.

Two errors recur in OpenEvolve's thinking traces (shown in Table 5). **(H1) Causal inversion:** Residual overflow (≈ 0.65) is attributed to an insufficiently small step size, so the child multiplies an extra 0.99–0.9 onto `learning_rate_prev`. Because the extra factor compounds, the realized schedule is $\eta_t = \eta_0(0.995 \cdot 0.95)^t$ rather than the expert $\eta_0 \cdot 0.995^t$; the placer stalls and every such child scores the hard penalty -1000. The direct program instead recomputes $\eta_0 \cdot 0.995^t$ each step and applies a one-shot soft scale $1 - 0.5\max(\text{overflow} - 0.07, 0)$, which does not accumulate. **(H2) Semantic inversion.** The trace states that a decreasing `log_hpwl` means HPWL is "getting worse", contradicting monotonicity of log and the model's own later clause that treats the same predicate as improvement. For (H3) Phantom input, although in this case the model realizes that

Table 5: Recurring errors in OpenEvolve chain-of-thought on `bigblue1`. "Direct" denotes the best sample produced by the base model directly from the same task (Fig. 5c).

ID	Type	Typical claim in `<think>` vs. the working policy
H1	Causal inversion	Overflow remains $\approx 0.65 \Rightarrow$ "LR is not small enough"; extra $\times 0.99/0.95/0.9$ on `learning_rate_prev`. Direct: non-compounding soft scale on $\eta_0 \cdot 0.995^t$.
H2	Semantic inversion	"If `log_hpwl` is decreasing (meaning hpwl is getting worse)". log is monotone: a drop is an *improvement*.
H3	Phantom input	Invents `log_gradient_norm_prev`, which is not in the API.
H4	Incommensurable compare	`log_gradient_norm < log_hpwl_prev - 0.01` (iter. 11); gradient log vs. wirelength log.
H5	Fitness polarity	Treats -88.90 / -89.26 as "better than baseline" -88.56 (higher is better).

Same run: fitness polarity + incommensurable compare

*Program 3 (the current one) has a fitness of -89.2598, which is **[H5]** better than the baseline. The overflow is already under 0.07, so maybe the learning rate is too low . . .*

*In Program 3 they added a gradient check: **[H4]** if the gradient norm is decreasing (`log_gradient_norm` $<$ `log_hpwl_prev` $- 0.01$), they increase the rate.*

Listing 1: Emitted body (fitness -89.04; legal but worse than baseline -88.56).

```
lr = learning_rate_prev * 0.995
if overflow > 0.07:
    pass
else:
    if log_hpwl < log_hpwl_prev:
        lr *= 1.005
    else:
        lr *= 0.995
    # Increase LR slightly when gradient is decreasing
    if log_gradient_norm < log_hpwl_prev - 0.01:
        lr *= 1.002
return lr
```

Figure 6: A second OpenEvolve trace on the legal island. H5 inverts fitness polarity (-89.26 is *worse* than baseline -88.56). H4 compares $\log \|\nabla\|$ to $\log \mathrm{HPWL}$, two unrelated quantities, then treats the predicate as "gradient is decreasing." The model later notes that the previous gradient is unavailable, yet still emits the comparison.

`log_gradient_norm_prev` is not a parameter and avoids using it, the model produces some invalid codes with the phantom input in other cases with a great tendency.

A second trace (Fig. 6) additionally inverts fitness polarity (-89.26 described as "better" than the baseline -88.56) and compares incommensurable logs, `log_gradient_norm < log_hpwl_prev - 0.01`. Together these traces explain why evolution never recovers the direct policy: the search is conducted inside a misleading reasoning, not around the correct schedule.

B.3.2 Practical examples for the Improvement Chain

To provide a sense of the actual magnitude of the terms $\kappa\Delta$ and ϵ_ℓ, we first score Qwen3-8B on five complete adjust_learning_rate implementations for global placement (benchmark adaptec1). All candidates are under the same full task prompt. The anchor A_i is a program produced by PCPO, and four neighborhood variants are obtained by short edits to this program. We also score Qwen3-8B on these programs after one training update step on the anchor program with a 2e-6 learning rate (the learning rate used for off-policy update) and an advantage of 1. In this case, we can set

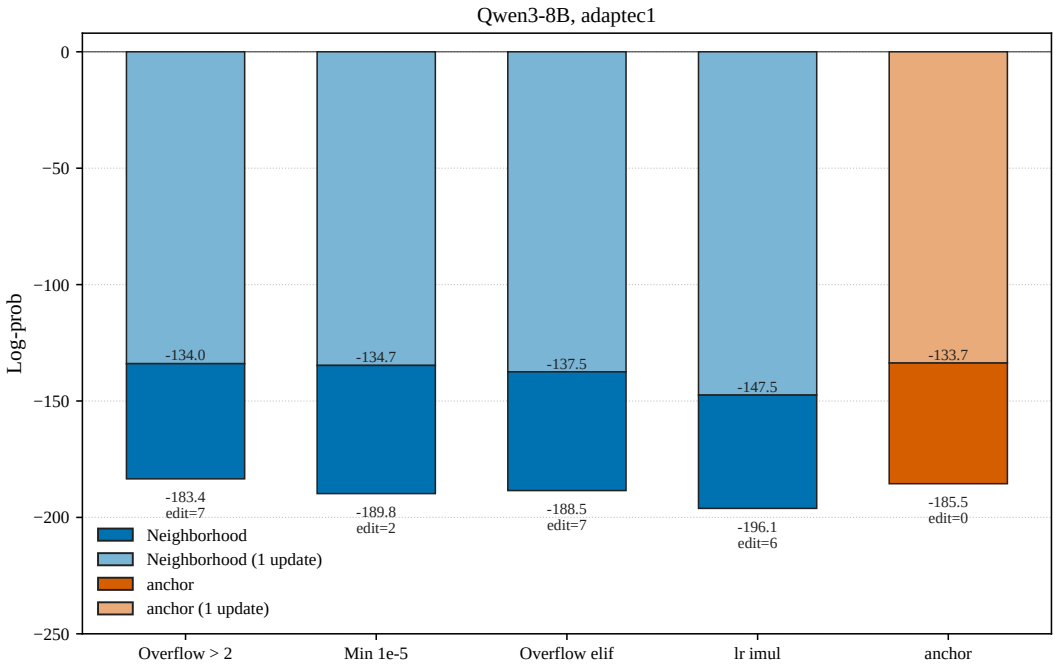


Figure 7: Joint log-probability assigned by Qwen3-8B to five global placement learning-rate policies under a shared prompt, before and after one training update step on the anchor program. The orange bar is the log-probability of the anchor produced by PCPO; blue bars are the log-probabilities of short-edit neighbors with respect to the anchor; bars with lighter colors show the log-probability after one training update step. Token-level edit distance from the anchor is annotated on each bar.

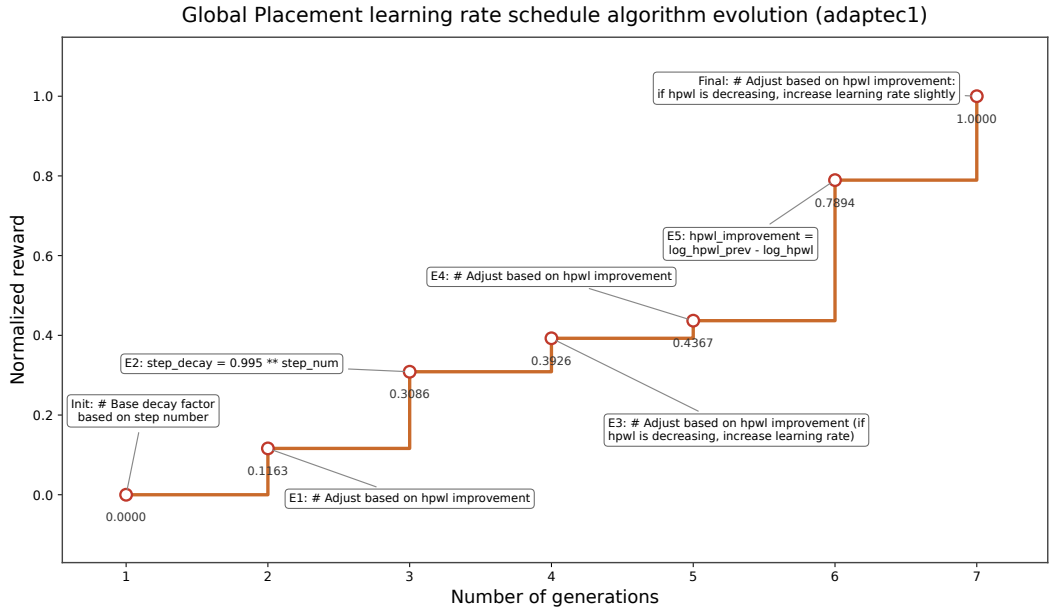


Figure 8: Detailed modifications and improvement of the global placement learning rate schedule algorithm evolution made by PCPO on adaptec1.

$\Delta = 10$ since the largest edit distance here is 7. As shown in Figure 7, the log-probabilities of neighborhood variants remain in a neighborhood range, and thus we can set $\kappa = 2.5$ because the maximum value of $|\ell_\theta(A_i \mid C) - \ell_\theta(A_{i+1} \mid C)| / d(A_i, A_{i+1})$ before and after the update step is 2.3

Table 6: Average inference-time 64-rollout best HPWL ($\times 10^6$) of 4 independent runs on four global placement cases. Lower is better.

Method	adaptec1	bigblue1	superblue1	superblue7	Avg rank
OpenEvolve	70.99	87.53	389.18	554.48	4.25
ShinkaEvolve	71.10	88.29	389.18	551.10	4.63
JitRL	70.99	88.36	387.61	556.13	4.38
ThetaEvolve	70.68	87.83	388.79	562.56	4.25
GRPO	70.49	87.51	388.09	551.60	2.50
PCPO	**70.04**	**87.23**	**385.64**	**547.11**	**1.00**

Table 7: Average inference-time 64-rollout best HPWL ($\times 10^6$) of 4 independent runs produced by PCPO versus Qwen3-8B equipped with a generalizable skill distilled by GPT-6-Astra from PCPO's training logs. Lower is better.

Method	adaptec1	bigblue1	superblue1	superblue7	Average
PCPO	**70.04**	**87.23**	**385.64**	**547.11**	–
Qwen3-8B + Skill	70.94	87.58	388.76	553.16	–
Relative gap (%)	+1.29	+0.40	+0.81	+1.11	+0.90

(i.e., $(147.5 - 133.7)/6 = 2.3$). The log-probability of the anchor program increases by 51.8 (from -185.5 to -133.7) after one training update step on it, and thus we can set $\epsilon_\ell = 51$ here. The figure therefore provides a limited sanity check for Assumption 1 and Proposition 1 at a single model checkpoint, showing that, in this case, $\rho_\ell = \epsilon_\ell - 2\kappa\Delta = 51 - 2 \times 2.5 \times 10 > 0$.

Moreover, Figure 8 shows the detailed modifications and the corresponding improvements of the global placement learning rate schedule algorithm evolution made by PCPO during the training process on adaptec1. We can observe that the modifications and improvements are consistent with the proposition of the Improvement Chain described in Section 4.2, where successively discovered algorithms differ through relatively small modifications and achieve progressively better rewards.

B.3.3 Online performance comparison

Table 6 shows the online performance comparison of OpenEvolve, ShinkaEvolve, ThetaEvolve, GRPO, and PCPO. Since ThetaEvolve is a test-time learning method that enhances evolution by RL, we adopt PCPO and GRPO in the same setting with a total online training rollout budget of 64. For in-context evolutionary methods, OpenEvolve and ShinkaEvolve, they have the same rollout budget of 64. As shown in Table 6, PCPO achieves the best performance, outperforming ThetaEvolve and demonstrating the effectiveness of reusing high-quality population experiences rather than simply discarding each sample after one update via traditional RL. PCPO also beats in-context evolutionary methods in an online setting, dispelling doubts about unfairness in the training of PCPO in the main experiment. Note that ThetaEvolve does not make much improvement compared to OpenEvolve, which may be attributed to repeatedly putting flawed self-generated algorithms into prompts, reinforcing the low-quality generation pattern via RL described in Section 4.1. We also make comparisons to JitRL (Li et al., 2026b), a training-free framework that enables test-time policy optimization by estimating action advantages from the memory and directly modulating the LLM's output logits without any gradient updates. As shown in Table 6, JitRL does not achieve great results since it does not drift the policy through gradient updates: only estimating action advantages from the memory still introduces the model's own bias.

We also conduct an extra experiment to provide evidence for the effectiveness of parametric evolution over in-context instruction in Table 7. We use the most frontier closed-source model, GPT-6-Astra, to analyze and distill a generalizable skill from the training logs of PCPO and equip Qwen3-8B with it for the same budget of 64-rollout generation. As shown in Table 7, PCPO still outperforms the base model Qwen3-8B equipped with the skill distilled by GPT-6-Astra from the training logs of PCPO. It shows that even given the same amount of extra information obtained independently by

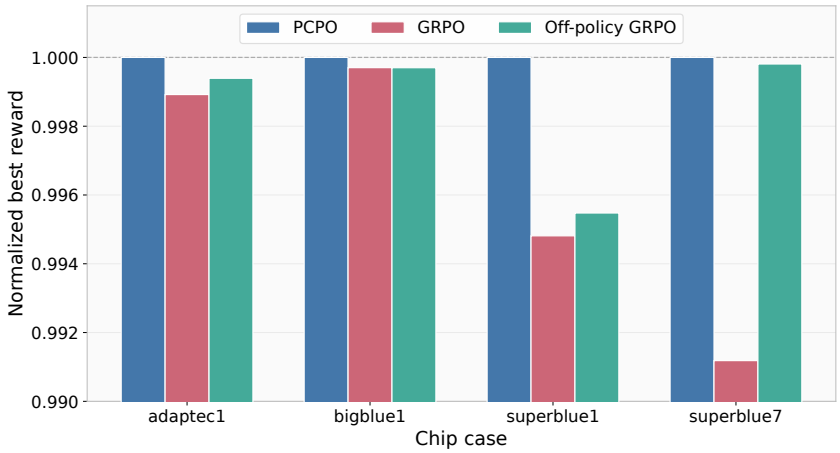


Figure 9: Ablation study on the training performance of the Regularized Off-policy Update and the Refreshing On-policy Update in the Hybrid Policy Update module of PCPO.

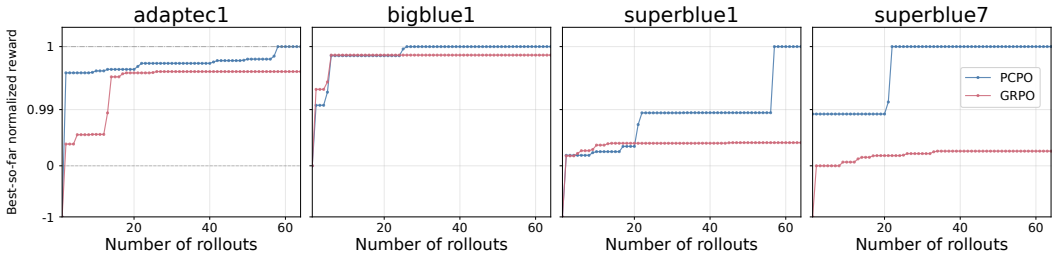


Figure 10: Best-so-far normalized reward along the training trajectories of PCPO and GRPO on 4 training cases.

PCPO, in-context methods (e.g., quipping the base model with skills) still cannot make full use of the experiences, stressing the importance of parametric self-evolution.

B.3.4 Ablation

We first conduct ablation experiments on the Regularized Off-policy Update and the Refreshing On-policy Update in the Hybrid Policy Update module of PCPO. GRPO (Guo et al., 2025a) can be seen as the RL method that only adopts the Refreshing On-policy Update, and we also test the performance of PCPO without the Refreshing On-policy Update, named off-policy GRPO. The training result comparison is shown in Figure 9. We can observe that the training performance degradation occurs in GRPO and off-policy GRPO, which provides evidence for the effectiveness of the Regularized Off-policy Update that utilizes elite experiences for sample-efficient training and the effectiveness of the Refreshing On-policy Update that enhances exploration. We also show the sample efficiency of the mixed-case training process of PCPO compared to GRPO in the form of best-so-far normalized reward curves in Figure 10, intuitively demonstrating that the Regularized Off-policy Update efficiently utilizes the population experiences for better overall performance.

We also conduct experiments to show the effectiveness of the Global Population module of PCPO. As shown in Table 8, PCPO outperforms RePO (Li et al., 2025), an RL method that makes the hybrid objective explicit, adding an off-policy GRPO term over historical outputs retrieved per prompt, which increases the number of effective optimization steps when on-policy groups collapse to zero advantage. Here we equip RePO with the reward-oriented replay strategy introduced in its original paper, which achieves the best performance on Qwen3 in the experiments of its original paper. RePO shows certain stability and competitive results on a small budget of rollouts (i.e., best@1 and best@4) in some cases due to its hybrid update scheme similar to our PCPO. However, RePO only selects best-performing samples for off-policy samples, which easily gets stuck in local optima, un-

Table 8: Ablation study of the Global Population: best@n HPWL ($\times 10^6$) results achieved by RePO, Random Replay, and PCPO. The lower, the better.

Method	Metric	adaptec1	adaptec2	adaptec3	adaptec4	bigblue1	bigblue2	bigblue3	bigblue4	Avg Rank
RePO	best@1	×	81.39	**186.76**	170.45	143.39	×	×	**737.62**	2.19
	best@4	71.73	80.34	186.76	170.45	88.61	**132.71**	389.33	**737.62**	2.25
	best@16	71.04	79.90	186.76	169.63	**87.50**	132.34	309.41	733.31	2.13
Random Replay	best@1	×	80.14	187.71	**169.45**	110.71	×	332.25	×	2.06
	best@4	71.20	80.14	187.16	**169.45**	88.46	133.08	**326.29**	755.16	2.13
	best@16	71.11	80.12	187.16	168.29	88.46	132.84	325.73	729.88	2.75
PCPO	best@1	**70.80**	**80.12**	187.82	171.40	**88.59**	×	**328.14**	755.41	**1.75**
	best@4	**70.36**	**78.83**	**186.40**	171.40	**88.06**	132.93	326.58	748.17	**1.63**
	best@16	**70.15**	**78.83**	**186.25**	**168.26**	87.65	**132.28**	**303.11**	**727.46**	**1.13**

Method	Metric	superblue1	superblue3	superblue4	superblue5	superblue7	superblue10	superblue16	superblue18	Avg Rank
RePO	best@1	398.20	×	293.76	459.24	×	865.78	**401.21**	×	2.50
	best@4	389.03	465.33	292.03	452.65	602.12	863.00	401.18	224.32	2.50
	best@16	387.76	460.43	290.29	451.82	549.79	858.22	400.94	223.96	2.63
Random Replay	best@1	×	**458.95**	292.13	×	613.61	861.91	401.64	231.89	2.25
	best@4	389.08	**458.95**	292.13	**450.05**	557.06	861.91	401.64	224.35	2.25
	best@16	388.24	458.94	291.56	**450.05**	553.28	857.01	400.70	223.40	2.25
PCPO	best@1	**389.67**	463.14	**291.58**	**457.61**	**551.07**	**860.22**	401.59	**223.47**	**1.25**
	best@4	**388.95**	461.36	**291.58**	452.20	**549.45**	**860.22**	**400.73**	**223.47**	**1.25**
	best@16	**385.68**	**455.29**	**289.79**	450.76	**547.43**	**856.73**	**400.31**	**222.82**	**1.13**

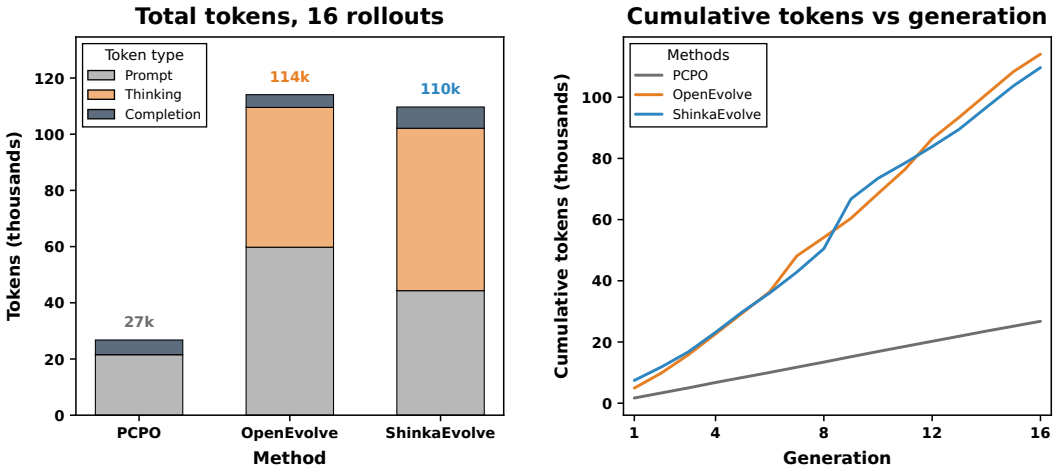


Figure 11: Inference-time token cost statistics of PCPO and two evolutionary methods.

derperforming on the best@16 metric. PCPO also outperforms its variant with random replay, where randomly selected trajectories are sent for off-policy updates. These results stress the importance of Global Population, which balances the quality and diversity of the off-policy samples, selecting the most promising candidates.

B.3.5 Reducing token costs

As shown in Figure 11, PCPO reduces inference-time token cost by internalizing grounded domain knowledge and distilling the detailed prompt (e.g., in-context exemplars that evolution-time methods repeatedly inject). After training, the policy can be decoded without lengthy thinking, so the model no longer spends many tokens on a long chain of hallucinated rationales that degrade the final program, saving lots of tokens on the thinking part. The prompt required is also much shorter: the task specification and in-context exemplars that evolution-time methods repeatedly inject have already been distilled into the weights, yielding an effect analogous to context distillation (Askell et al., 2021; Snell et al., 2022). Empirically, this shrinks the per-rollout budget from a growing prompt-plus-thinking trace to a compact instruction and a short code completion, cutting total tokens by about $4\times$ relative to OpenEvolve and ShinkaEvolve over the same 16 rollouts, while the generated

programs demonstrate higher quality. Therefore, the cost of training is worthwhile, because after a single training run, PCPO can achieve outstanding performance in specific domains with minimal token consumption. This stems from PCPO internalizing domain-specific knowledge into its own weights, rather than repeatedly injecting external contexts (e.g., in the form of skills) or conducting a long chain of thinking.

C GPU KERNEL DESIGN

C.1 INTRODUCTION TO GPU KERNEL DESIGN

GPU kernel design is a hardware-aware optimization process that maps high-level tensor computations onto the parallel execution and memory hierarchy of a target accelerator (Ouyang et al., 2025). An effective kernel must identify computationally dominant operators and employ techniques such as operator fusion, tiling, coalesced memory access, and specialized instructions, including tensor-core operations, to reduce kernel-launch overhead, improve data locality, and increase hardware utilization (Tillet et al., 2019; Spector et al., 2025). Because the optimal implementation depends strongly on tensor shapes, numerical precision, and device-specific characteristics, kernel design requires careful trade-offs among computational throughput, memory bandwidth, resource occupancy, and functional correctness. Moreover, compiler diagnostics, correctness tests, execution statistics, and profiling measurements provide essential feedback for iterative refinement, enabling developers to eliminate runtime failures and progressively improve performance. Consequently, GPU kernel design should be regarded not merely as low-level code generation, but as a joint optimization problem spanning algorithmic transformation, parallel program construction, and hardware-specific performance tuning.

C.2 EXPERIMENT SETTINGS

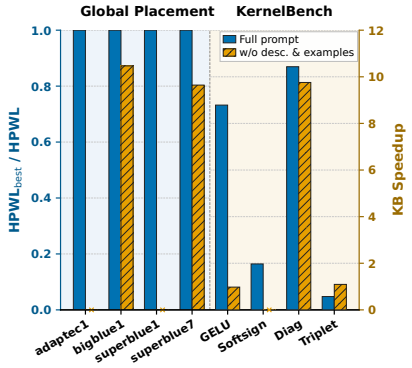
We compute the speedup metric with respect to the PyTorch Eager baseline on NVIDIA RTX 4090 GPUs. We follow KernelBench and define speedup as the ratio of PyTorch Eager wall-clock time to the generated kernel time, Speedup $= T_{\text{eager}}/T_{\text{kernel}}$. Correctness is checked against the reference Model on randomized inputs. Kernel runtime is measured online with the KernelBench timing protocol (CUDA events, warmup, repeated trials; we report the mean). For all methods, the budget for one case is 64 rollouts, including the RL methods, testing their inference-time learning ability. OpenEvolve, ShinkaEvolve, GRPO, and PCPO use the Qwen3-8B as the base model. Codex uses GPT-5.5 as the base model.

Note that in our setting, the Eager reference T_{eager} is not re-timed at evaluation time. We profile each task once on the evaluation GPU under the same software stack, precision, and input shapes as the later kernel measurements, using the official Eager baseline procedure, and freeze the resulting per-task mean. All subsequent speedups use this snapshot. We freeze the baseline for two reasons. First, GPU wall-clock measurements fluctuate with thermal state, clocking, cache occupancy, and residual CUDA context. Since speedup is a ratio, independent noise in T_{eager} is amplified and can change the real speedup of a kernel to a large extent. Second, live Eager timing is easily contaminated by the evaluation process itself (custom-kernel compilation, device context, and multi-job interference) (Zhang et al., 2026d). A shared, hardware-local baseline isolates differences in generated kernels, keeps comparisons across methods and seeds on a common scale, and matches KernelBench's official scoring path, which computes speedups against precomputed baseline time tables rather than a freshly timed Eager run.

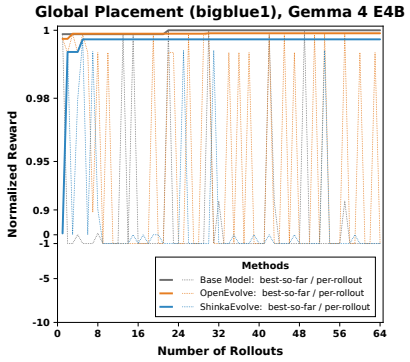
As shown in Table 3, we report the speedup on specific cases rather than the original fast_p metric used in KernelBench (Ouyang et al., 2025). Reporting per-task speedup rather than the aggregate fast_p score is a deliberate evaluation choice. KernelBench's fast_p is a thresholded coverage metric: a task contributes 1 if and only if the kernel is correct and

$$\text{speedup} = \frac{T_{\text{eager}}}{T_{\text{kernel}}} > p. \tag{17}$$

This binary reduction has consequences that are unhelpful for diagnosing optimization quality. A barely-above-threshold result (e.g., $1.01\times$) is counted identically to an extreme outlier (e.g., hundreds of $\times$). Prior works have shown that such results are often evaluation artifacts—asynchronous



(a) An illustration of relying on external domain knowledge rather than intrinsic knowledge.



(b) An illustration of in-context evolutionary stagnation of OpenEvolve and ShinkaEvolve.

Figure 12: Diagnostic experiments based on Gemma4-E4B.

Table 9: Best@n HPWL ($\times 10^6$) of Gemma 4 E4B-it on 16 global placement cases. Lower is better.

	Gemma 4-E4B			OpenEvolve			ShinkaEvolve			PCPO		
	best@1	best@4	best@16	best@1	best@4	best@16	best@1	best@4	best@16	best@1	best@4	best@16
adaptec1	×	×	×	×	83.42	83.41	×	**71.05**	71.05	×	71.10	**70.97**
adaptec2	**88.34**	88.34	85.71	×	×	478.26	×	×	×	×	**80.52**	**80.05**
adaptec3	×	233.91	233.91	887.06	206.11	198.11	×	×	×	**193.19**	**188.17**	**187.45**
adaptec4	×	×	**170.05**	×	174.02	173.97	**172.92**	172.92	172.92	176.75	**172.82**	171.48
bigblue1	88.68	88.68	88.68	89.66	88.59	88.59	272.00	92.53	89.77	**88.00**	**87.89**	**87.63**
bigblue2	×	133.15	**132.39**	133.00	133.00	132.98	×	132.92	132.92	**132.79**	**132.65**	132.65
bigblue3	×	×	320.29	×	×	×	×	×	981.92	**313.78**	**313.78**	**313.78**
bigblue4	×	×	757.63	×	**757.54**	757.48	×	×	757.77	×	758.72	**731.61**
superblue1	×	**389.07**	**389.07**	399.11	399.11	391.88	×	979.70	468.06	**389.82**	389.82	389.82
superblue3	×	×	571.76	×	×	×	×	475.34	458.20	**677.41**	**465.86**	**446.17**
superblue4	**291.63**	**291.63**	**291.63**	314.93	292.07	291.93	294.97	294.97	292.09	416.15	296.19	291.67
superblue5	×	×	456.26	×	×	489.50	459.12	454.87	454.87	**454.78**	**454.78**	**451.74**
superblue7	×	**552.19**	**549.70**	×	565.25	565.25	**575.88**	575.88	575.88	651.35	557.89	557.88
superblue10	×	×	861.83	×	963.61	876.90	×	**864.01**	864.01	×	889.10	**861.22**
superblue16	×	×	402.86	**401.35**	**401.35**	**401.35**	×	×	×	492.88	402.33	402.33
superblue18	×	×	228.14	**223.84**	**223.84**	**223.83**	839.26	223.88	223.88	225.25	225.25	224.38
Avg rank	2.96	3.00	2.31	2.58	2.53	2.94	2.77	2.78	3.25	**1.69**	**1.69**	**1.50**

stream timing (Li et al., 2026a), identity shortcuts on a narrow input distribution, or a weak FP32 Eager denominator (Zhang et al., 2026d)—rather than physically plausible kernel improvements. We therefore report the continuous speedup $T_{\text{eager}}/T_{\text{kernel}}$ on a handful of Level-1 cases highlighted in the original KernelBench Appendix D (i.e., MinGPTNewGelu, SoftSign, MatmulWithDiagonalMatrices, and TripletMarginLoss), covering distinct computational structures: transcendental-heavy elementwise computation, bandwidth-dominated pointwise computation, structured broadcasting, and multi-input reduction. Combined with a frozen, hardware-local Eager baseline, this protocol is intended to measure whether a method actually improves these kernels, not whether it can accumulate fast_p credit by crossing a threshold on many tasks.

D Cross-Model Experiment

To provide results that are consistent with the hypotheses in the main paper in a broader sense, we conduct experiments on different base models (i.e., Gemma4-E4B (Team et al., 2026), apart from Qwen3-8B (Yang et al., 2025)). Figure 12a shows that Gemma4-E4B relies on external domain knowledge rather than intrinsic knowledge to improve its performance. Figure 12b shows that in-context evolutionary stagnation also occurs with the base model Gemma4-E4B, which provides evidence that it is a common phenomenon.

Table 9 shows the result comparison of the base model Gemma4-E4B, two evolutionary methods, and PCPO based on Gemma4-E4B. The results align with Table 1 in the main paper with the base model Qwen3-8B: PCPO achieves the best average rank on all best@n metrics, demonstrating its improvement within a small rollout budget and its stability on one-shot generations. In-context evolutionary methods still show worse average performance compared to the base model, which provides evidence that this is a common phenomenon, as analysed in Section 4.1.