

EvoX: Meta-Evolution for Automated Discovery

Shu Liu^{1,*}, Shubham Agarwal^{1,*}, Monishwaran Maheswaran¹, Mert Cemri¹
 Qiuyang Mang¹, Zhifei Li¹, Ashwin Naren¹, Ethan Boneh², Audrey Cheng¹
 Alexander Du¹, Melissa Pan¹, Kurt Keutzer¹, Alvin Cheung¹, Koushik Sen¹
 Alex Dimakis^{1,3}, Matei Zaharia¹, Ion Stoica¹

¹University of California, Berkeley ²Stanford University ³Bespoke Labs

*Equal contribution.

Abstract

Recent work such as AlphaEvolve has shown that combining LLM-driven optimization with evolutionary search can effectively improve programs, prompts, and algorithms across domains. In this paradigm, previously evaluated solutions are reused to guide the model toward new candidate solutions. Crucially, the effectiveness of this evolution process depends on the search strategy: how prior solutions are selected and varied to generate new candidates. However, most existing methods rely on fixed search strategies with predefined knobs (e.g., explore-exploit ratios) that remain static throughout execution. While effective in some settings, these approaches often fail to adapt across tasks, or even within the same task as the search space changes over time.

We introduce EvoX, an adaptive evolution method that optimizes its own evolution process. EvoX jointly evolves candidate solutions and the search strategies used to generate them, continuously updating how prior solutions are selected and varied based on progress. This enables the system to dynamically shift between different strategies during the optimization process. Across nearly 200 real-world optimization tasks, EvoX consistently outperforms prior open frameworks, including OpenEvolve, ShinkaEvolve, and GEPA. It achieves the best open-source performance on FrontierCS (+34% median over the best AI solutions across 172 problems), matches or surpasses AlphaEvolve and human-designed state-of-the-art solutions on multiple benchmarks, and does so at more than $3\times$ lower cost on average compared to existing open systems and Claude Code. Our code is available at github.com/skydiscover-ai/skydiscover.

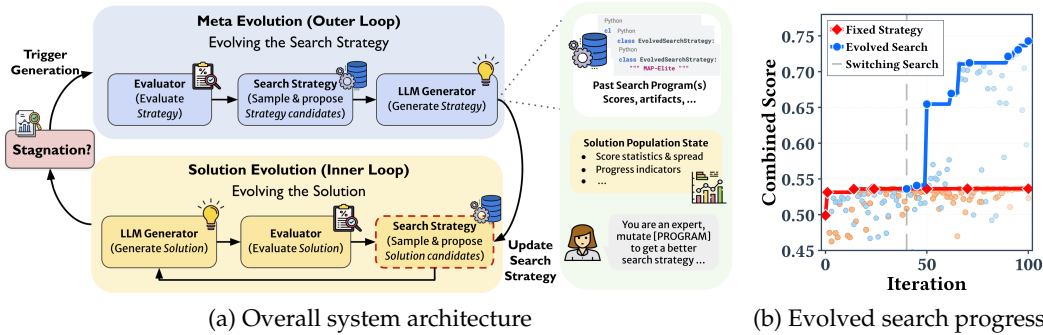


Figure 1: **Evolving the search strategy.** (a) *System architecture.* EvoX has two coupled loops: an inner loop that evolves solutions, and an outer loop that evolves the *search strategy* that governs generation. (b) *Effect of search evolution.* A strategy with fixed exploration-exploitation ratio (MAP-Elites, red) stagnates, while an evolving search strategy (blue) produces discrete performance breakthroughs.

1 Introduction

LLM-driven optimization combined with evolutionary search has enabled numerous scientific breakthroughs across domains like mathematics (Novikov et al. (2025)), systems performance optimization (Cheng et al. (2025a)), and competitive programming (Imajuku et al. (2025)). Typically, LLM-driven evolutionary systems maintain a population of candidate solutions. At each step, a *search strategy* selects a subset of previously evaluated solutions and constructs a prompt, which is passed to a generator model (typically an LLM) to produce new candidates. These candidates are then evaluated and added back to the population.

Critically, the effectiveness of the evolution process relies on the search strategy, which determines both (i) which candidates are selected from the population and (ii) how new solutions are proposed from them (e.g., via refinement, structural variation, or combining multiple ideas). These choices directly influence what the generator attempts next and which regions of the solution space are explored.

Existing LLM-driven evolutionary systems often rely on fixed search strategies with hand-specified parameters. For example, AlphaEvolve (Novikov et al. (2025)) employs MAP-Elites with predefined population database structures and selection ratios. OpenEvolve (Sharma (2025)) similarly relies on static elite and diversity heuristics. ShinkaEvolve (Lange et al. (2025)) takes a step toward adaptivity by incorporating bandit-based selection on LLM generators. However, key search strategy knobs remain manually configured. For example, the exploitation ratio is fixed, controlling how often top solutions are refined.

In practice, a fixed search strategy often fails to generalize across problems or across different stages of optimization. This can lead to stagnation during the search and require manual retuning to make progress. Some tasks benefit primarily from repeated refinement of a strong candidate, where small local modifications incrementally improve an existing solution. For example, in packing problems (Novikov et al. (2025); Georgiev et al. (2025)), adjusting a few placements in a near-feasible configuration can yield further gains. In contrast, other tasks require qualitatively different solution structures. For instance, in a GPU-model placement problem, refining a least-loaded heuristic quickly plateaus, and further progress requires reformulating the problem as a bin-packing assignment.

Even within a single optimization run, the effectiveness of a search strategy can change over time. In a multi-objective signal processing task (Figure 1(b)), a MAP-Elites style strategy (red line) makes rapid early progress but later stagnates. Switching to a strategy that explicitly samples candidates along different trade-off objectives (blue line) enables continued improvement.

Motivated by this observation, EvoX frames LLM-driven optimization as a meta-learning problem in which *the search strategy itself is treated as an evolvable object*. Rather than fixing a hand-tuned search strategy, EvoX operates as a two-level evolution process comprising a *solution-evolution* loop and a *meta-evolution* loop (Figure 1(a)).

The solution-evolution loop generates candidate solutions under the standard LLM-driven evolutionary search paradigm. At each step, a search strategy selects prior candidates and determines how to generate new ones, for example by sampling high-performing solutions and applying a variation operator (e.g., refinement or structural variation). The resulting prompt is passed to the LLM, which produces a new candidate that is evaluated and added back to the population.

The meta-evolution loop periodically updates the search strategy. Each strategy is deployed for a window of solution-evolution iterations and evaluated by the progress it induces on the downstream task (e.g., improvement rate). When progress stagnates, the meta-evolution loop generates a new search strategy using the LLM, conditioned on prior strategies, their observed performance, and the current state of the solution population. This loop enables EvoX to evolve the search strategy itself: selecting, mutating, and replacing strategies based on their effectiveness. Because the performance of a strategy depends on the evolving population, EvoX explicitly conditions strategy generation on population-level signals, allowing it to adapt as the search space changes.

Contributions. We evaluate EvoX across nearly 200 real-world optimization tasks spanning mathematics, algorithms, and scientific benchmarks (Table 1, Section 5). Starting from a simple random sampling policy, EvoX consistently outperforms prior LLM-driven evolutionary frameworks (OpenEvolve, ShinkaEvolve, GEPA) on the majority of tasks.

EvoX matches or surpasses AlphaEvolve on 5/7 math benchmarks and surpasses prior human-best solutions on 6/6 systems performance tasks. On the FrontierCS benchmark (172 programming challenges Mang et al. (2025)), EvoX achieves the best open-source performance, improving median scores by 34% over prior AI systems. Finally, EvoX is cost-efficient, breaking optimization plateaus for less than \$10 on average, while other systems and Claude Code typically require $\sim 3\times$ higher cost and often stagnate.

In summary, we make the following contributions:

1. We formalize LLM-driven optimization as a two-level process that separates solution evolution from search strategy evolution.
2. We introduce EvoX, a method that dynamically evolves search strategies based on improvements in the optimization objective.
3. We demonstrate consistent improvements over prior methods across nearly 200 problems, and characterize the cost, scaling behavior, and adaptation dynamics of search strategy evolution, including substantial gains in benchmark performance, real-world systems metrics, and cost efficiency.

2 Related Work

Context and Memory Management. Many LLM systems iteratively refine solutions using feedback from prior attempts, making context management critical. ACE (Zhang et al. (2025)), MemGPT (Packer et al. (2023)), Reflexion (Shinn et al. (2023)), and Self-Refine (Madaan et al. (2023)) improve how past information is stored and reused. Evolutionary search builds on this by treating prior solutions as a population for selection and variation.

LLM-Guided Evolutionary Search. Recent work applies evolutionary search to guide LLM-driven optimization. These systems differ primarily in how candidates are selected and varied. AlphaEvolve (Novikov et al. (2025)) selects candidates using MAP-Elites, GEPA (Agrawal et al. (2025)) selects along Pareto frontiers, and OpenEvolve (Sharma (2025)) and ShinkaEvolve (Lange et al. (2025)) emphasize diverse selection. CodeEvolve (Assumpção et al. (2025)) integrates LLM-based generation within an island-based genetic algorithm, while DeltaEvolve (Jiang et al. (2026)) models semantic deltas between candidates. PACEvolve (Yan et al. (2026)) introduces mechanisms such as hierarchical context pruning to improve stability. However, these systems still operate under predefined search strategies with manually designed knobs. Learning-based approaches such as ThetaEvolve (Wang et al. (2025)), TTT-Discover (Yuksekgonul et al. (2026)), and FLEX (Cai et al. (2025)) apply reinforcement learning to improve the generator model, but not the search process itself.

AdaEvolve (Cemri et al., 2026) improves search adaptability by tuning search-control parameters (e.g., exploration intensity and compute allocation) based on an improvement signal, achieving stronger cost-efficiency and rapid early progress at low iteration budgets. However, it relies on human-designed adaptation rules. In contrast, EvoX evolves the entire search strategy, enabling better scalability over longer horizons (Appendix A.3).

Meta-Learning and Learning to Optimize. Meta-learning and learned optimization frameworks treat the optimization procedure itself as the object of adaptation rather than a fixed component (Metz et al. (2019); Chen et al. (2022)). Prior work explores learned optimizers, symbolic discovery of update rules (Chen et al. (2023)), gradient-based meta-learning (Andrychowicz et al. (2016)), and reinforcement learning-based optimizers (Qu et al. (2025)). These works demonstrate how adapting the optimization process can significantly improve search efficiency. Our work extends this principle to LLM-driven evolutionary search.

More closely related to our setting, STOP and MoH learn optimizers offline before applying them to target tasks (Zelikman et al., 2024; Shi et al., 2026), whereas EvoX co-evolves

solutions and search strategies *online* within each target run. EoH and LMEA evolve candidates under predefined outer-loop procedures (Liu et al., 2024a;b). Although EoH uses multiple prompt variants and LMEA adapts the sampling temperature, neither adapts search-control decisions such as parent selection or population-context construction, which EvoX evolves directly. Appendix B.1 provides controlled comparisons with these methods.

EvoX: Meta-Evolving the Search Strategy. Building on these directions, EvoX treats the search strategy as an evolvable object and adapts it online based on feedback. This enables the search process (i.e., not only its candidate solutions or individual control parameters) to change across optimization stages and solution landscapes.

3 Problem Formulation

We formalize LLM-driven evolutionary search following standard abstractions from prior work (Romera-Paredes et al. (2024); Novikov et al. (2025); Sharma (2025); Agrawal et al. (2025); Lange et al. (2025)). Candidate solutions are generated by a language model, evaluated by a task-specific evaluator, and iteratively evolved under a search strategy and a fixed evaluation budget.

Candidate solution evaluation. Let $\mathcal{X}$ denote the space of candidate solutions (e.g., programs or prompts). Each candidate $x \in \mathcal{X}$ is evaluated by an evaluator $E(x) \rightarrow (s(x), a(x))$, which returns a scalar score $s(x) \in \mathbb{R}$ together with auxiliary artifacts $a(x)$ (e.g., logs, traces, or any feedback).

Solution population database. The optimization proceeds for T sequential evaluation steps. Let $\mathcal{D}_t = \{(x_i, s_i, a_i)\}_{i=1}^t$, $\mathcal{D}_0 = \emptyset$, denote the database of all candidate solutions evaluated up to step t . At each step, a new candidate is generated, evaluated, and appended to the database to form $\mathcal{D}_{t+1}$.

Search strategy. A search strategy $S \in \mathcal{S}$ specifies how the next-generation LLM input is constructed from the current database $\mathcal{D}_t$. Concretely, S defines the construction of the generation context: $C_S(\mathcal{D}_t) \rightarrow (x_{\text{par}}, \pi, \mathcal{I})$, which selects (i) one or more *parent* candidate(s) $x_{\text{par}} \in \mathcal{D}_t$ to be modified, (ii) a *variation operator* π expressed in the prompt that specifies how the parent can be modified, and (iii) optionally, an *inspiration set* $\mathcal{I} \subseteq \mathcal{D}_t$ (e.g., diverse candidates) that provides additional exemplars. This abstraction aligns with prior LLM-driven evolutionary systems (Lange et al. (2025); Sharma (2025)).

Given $(x_{\text{par}}, \pi, \mathcal{I})$, the solution generator (implemented by an LLM) produces a new candidate $x' \sim \mathcal{G}_{\text{sol}}(\cdot \mid x_{\text{par}}, \pi, \mathcal{I})$ which is evaluated and appended to the database.

Variation operators. The operator π specifies the *kind* of modification requested in a generation step. We use three variation operators: ① **local refinement** for fine-grained edits (exploitation), ② **structural variation** for coarse-grained redesigns (exploration), and ③ **free-form variation**, which imposes no constraints on the edit scope. These operators provide a simple, interpretable interface for controlling the explore-exploit tradeoff.

The semantics of these operators are task-dependent. For example, refinement may make localized edits (e.g., parameter tuning or code changes), while structural variation may alter the overall design (e.g., switching algorithm families). We instantiate operator-specific prompts once per problem using a lightweight model (Section 5), and the search strategy selects among them during optimization to control the type of variation applied.

While EvoX uses a fixed set of high-level operators for simplicity, it is not restricted to them. Appendix A.4 studies a variant that co-evolves operators (i.e., variation prompts), enabling task-specific discovery at the cost of slower convergence due to the larger search space.

Optimization goal. Search strategies shape optimization behavior, and our goal is to *adaptively select and improve the search strategy* to maximize the final best score $\max_{(x,s,a) \in \mathcal{D}_T} s$ under a fixed evaluation budget of T steps.

Algorithm 1 EvoX: Two-level evolution process

Require: Budget T , window W , stagnation threshold τ
Require: Evaluator E ; solution generator $\mathcal{G}_{\text{sol}}$; strategy generator $\mathcal{G}_{\text{str}}$
Require: Population descriptor $\phi(\cdot)$; validity test $\text{VALID}(\cdot)$
Require: Initial solution database $\mathcal{D}_0$; initial strategy S_0

```

1:  $\mathcal{D}_t \leftarrow \mathcal{D}_0$ ;  $S_t \leftarrow S_0$ ;  $\mathcal{H} \leftarrow \emptyset$ ;  $t \leftarrow 0$ 
2: while  $t < T$  do
  Phase I: Solution evolution under  $S_t$  (one window)
3:    $\phi_t \leftarrow \phi(\mathcal{D}_t)$ 
4:    $s_{\text{start}} \leftarrow \max_{(x,s,a) \in \mathcal{D}_t} s$ 
5:   for  $i = 1$  to  $W$  do
6:     if  $t \geq T$  then break
7:     end if
8:      $(x_{\text{par}}, \pi, \mathcal{I}) \sim C_{S_t}(\mathcal{D}_t)$ 
9:      $x' \sim \mathcal{G}_{\text{sol}}(\cdot \mid x_{\text{par}}, \pi, \mathcal{I})$ 
10:     $(s', a') \leftarrow E(x')$ 
11:     $\mathcal{D}_{t+1} \leftarrow \mathcal{D}_t \cup \{(x', s', a')\}$ ;  $t \leftarrow t + 1$ ;  $\mathcal{D}_t \leftarrow \mathcal{D}_{t+1}$ 
12:   end for
  Phase II: Progress monitoring
13:    $s_{\text{end}} \leftarrow \max_{(x,s,a) \in \mathcal{D}_t} s$ 
14:    $\Delta \leftarrow s_{\text{end}} - s_{\text{start}}$ 
15:    $J_t \leftarrow \Delta \log(1 + s_{\text{start}}) / \sqrt{W}$ 
16:    $\mathcal{H} \leftarrow \mathcal{H} \cup \{(S_t, \phi_t, J_t)\}$ 
  Phase III: Strategy evolution (on stagnation)
17:   if  $\Delta < \tau$  then
18:      $S' \sim \mathcal{G}_{\text{str}}(\cdot \mid \mathcal{H}, \phi(\mathcal{D}_t))$ 
19:     if  $\text{VALID}(S')$  then  $S_t \leftarrow S'$ 
20:     end if
21:   end if
22: end while
23: return  $\arg \max_{(x,s,a) \in \mathcal{D}_T} s$ 

```

4 Co-evolving Solution and Search Strategy

EvoX addresses the problem of iteratively improving solution quality under a fixed evaluation budget defined in Section 3 by *co-evolving* two components: (i) the solution population $\mathcal{D}_t$, and (ii) the search strategy S that constructs the next-generation LLM context. Specifically, EvoX proceeds through the three-step process shown in Algorithm 1: (1) evolving the solution database under a fixed search strategy, (2) monitoring population performance, and (3) updating the search strategy when progress stalls, using feedback from previous strategy deployments and the current state of the solution population.

Solution evolution under the current strategy. Given the current database $\mathcal{D}_t$ and an active strategy S_t , EvoX generates and evaluates new candidates as defined in Section 3: the strategy constructs a generation context $(x_{\text{par}}, \pi, \mathcal{I}) \sim C_{S_t}(\mathcal{D}_t)$, the generator produces x' , and the evaluator appends (x', s', a') to the database.

A search strategy controls both *what* the model sees (through parent selection and inspiration construction) and *how* the selected parent is transformed (via the variation operator π).

Progress monitoring and strategy updates. A search strategy does not affect a single candidate in isolation, but shapes a *sequence* of generated candidates. Accordingly, its effectiveness can only be assessed over multiple evaluation steps rather than from a single outcome. For this reason, EvoX monitors progress over a sliding window of the most recent W evaluation steps.

Let t denote the start of the current monitoring window and define

$$s_{\text{start}} = \max_{(x,s,a) \in \mathcal{D}_t} s, \quad s_{\text{end}} = \max_{(x,s,a) \in \mathcal{D}_{t+W}} s, \quad \Delta = s_{\text{end}} - s_{\text{start}}. \quad (1)$$

We treat Δ as the primary signal of strategy efficacy. If Δ falls below a stagnation threshold τ , EvoX triggers a strategy update; otherwise, it continues with the current strategy. This design makes strategy switching *demand-driven* rather than *periodic at a fixed interval*, avoiding unnecessary updates when the current strategy is effective.

Search strategy evaluation. After W evaluation steps have elapsed, EvoX computes a performance score for the strategy employed during that window:

$$J(S_t | \mathcal{D}_t) = \frac{(s_{\text{end}} - s_{\text{start}}) \log(1 + s_{\text{start}})}{\sqrt{W}} \quad (2)$$

The $\log(1 + s_{\text{start}})$ term upweights improvements achieved from higher starting scores, rewarding strategies that drive progress near the frontier where gains are typically harder to obtain. The $\sqrt{W}$ normalization accounts for window length. Appendix B.6 ablates each component of this score.

4.1 Meta-evolving the search strategy

Adapting search strategies is challenging because their effectiveness is *state-dependent*. For instance, a strategy that drives rapid progress at one stage of search may become ineffective as the solution population evolves. This arises from the inherently non-stationary nature of evolutionary optimization: as the database grows, the distribution of candidate quality, diversity, and variation outcomes shifts, altering which search behaviors are beneficial.

EvoX addresses this challenge by conditioning strategy updates on both *a population of past search strategies* and *the current downstream solution population state*. Conditioning on the population descriptor $\phi(\mathcal{D}_t)$ allows the system to interpret stagnation relative to population structure (e.g., loss of diversity, repeated parent selection, ineffective variation). Conditioning on the strategy database $\mathcal{H}$ provides empirical evidence about which strategies previously induced progress under similar states. Together, these signals enable state-conditional strategy adaptation rather than fixed or periodic strategy switching.

Search strategy database. To achieve this, EvoX maintains a *search strategy database* $\mathcal{H} = \{(S_j, \phi_j, J_j)\}_{j=1}^M$, which serves as a memory of previously deployed strategies. Each entry records a strategy S_j , a descriptor $\phi_j = \phi(\mathcal{D}_{t_j})$ summarizing the population state in which it was deployed, and its observed performance J_j computed using Eq. 2. This database enables EvoX to reason about *which strategies tend to work under which search conditions*.

Population state descriptor. The descriptor $\phi(\mathcal{D}_t)$ summarizes the current state of the solution population. It includes: (i) score statistics (best value, percentiles, spread), (ii) frontier structure (e.g., top- k scores), (iii) progress indicators (e.g., steps since last significant improvement), and (iv) recent window statistics (e.g., parent selection frequency).

Meta-evolution of search strategies. When a strategy update is triggered, EvoX evolves a new search strategy by applying variation to high-performing strategies from the history of the search. The strategy database $\mathcal{H}$ plays the role of a population, where each individual strategy S_j is associated with a score signal J_j evaluated in the population state ϕ_j in which it was deployed.

EvoX performs score-biased selection over $\mathcal{H}$ to choose a high-performing parent strategy to mutate, and selects an inspirational set of strategies prioritizing those that previously induced strong progress and those that performed well under similar population descriptors. The strategy generator (i.e., an LLM), denoted $\mathcal{G}_{\text{str}}$, then applies mutation to the selected parent, conditioned on the current population descriptor $\phi(\mathcal{D}_t)$, to produce a new strategy candidate S' : $S' \sim \mathcal{G}_{\text{str}}(\cdot | S_{\text{par}}, \phi(\mathcal{D}_t))$. Mutations modify the components of a strategy (e.g., parent selection rules, construction of the inspiration set, or preferences over variation operators π). EvoX performs safety validation (e.g., API and functional checks) for each candidate strategy before deployment, switching to it if valid or retaining the previous one.

otherwise, while preserving the existing solution population and continuing the search from the current state.

Here, “mutation” denotes an LLM-driven rewrite of the complete search-policy program rather than a classical fixed-length mutation. A rejected rewrite is not deployed, and the current valid strategy remains active until the next stagnation check. Appendix B.5 reports the observed validity rate of generated strategies and the size and types of learned program changes.

5 Evaluation

Benchmarks. We evaluate EvoX on 196 real-world optimization tasks spanning mathematics (8), systems (6), and algorithmic and research problems (10 from ALE-Bench-Lite (Imajuku et al. (2025)) and 172 from FrontierCS (Mang et al. (2025))) (Sections 5.1–5.2). We also report results on ARC-AGI-2 in Appendix A.7. Benchmark details are in Appendix E, and baseline framework configurations are in Appendix G.

Baselines and Setups. We compare EvoX against strong LLM-driven evolutionary frameworks, including OpenEvolve, ShinkaEvolve, and GEPA. For math tasks, we report AlphaEvolve results, and for both math and ADRS systems benchmarks, we include human-best results. Additional comparisons to Claude Code (Anthropic (2026)), CodeEvolve (Assumpção et al. (2025)), ThetaEvolve (Wang et al. (2025)), and concurrent work such as HyperAgents (Zhang et al. (2026b)) are provided in Appendix A.

For EvoX, we use GPT-5 for search strategy generation, with a stagnation window of 10% of the iteration budget to trigger strategy evolution (Section 4). All runs start from a *random search strategy* that samples parents and inspirations uniformly; other initial strategies (e.g., Best-of-N, MAP-Elites) are evaluated in Section 5.3, with full results in Appendix A.3 and A.6.

All *open* frameworks are evaluated under a fixed budget of 100 iterations per task unless stated otherwise. Complete equal-call token, dollar-cost, and wall-clock accounting is reported in Appendix B.2. AlphaEvolve (Novikov et al. (2025)) results are taken from the original paper, as it is not open-sourced and its iteration budget is unavailable. We report mean and best performance over three runs using GPT-5 and Gemini-3.0-Pro. Mean reflects robustness, while best captures peak performance; arrows ($\uparrow$ / $\downarrow$) indicate the optimization direction. Ablations on cost and scaling are presented in Section 5.3.

5.1 Main Results: Math Optimization Problems

We evaluate eight mathematical optimization tasks (continuous and discrete) spanning geometric packing, extremal geometry, distance maximization, and sequence design (Table 1). Across these tasks, **EvoX achieves the strongest overall performance among open evolutionary frameworks, including OpenEvolve, ShinkaEvolve, and GEPA.** Under GPT-5, EvoX attains the *best* or tied-best result on 7 of 8 tasks, and under Gemini-3.0-Pro, it achieves the best result on all 8 tasks. In terms of *mean* performance, EvoX achieves the best score on 6 of 8 tasks under GPT-5 and 7 of 8 tasks under Gemini-3.0-Pro, demonstrating robust performance across runs. In the two cases where mean performance lags best performance (Heilbronn triangle and MinMaxMinDist $d = 3$), early random initializations can converge to strong local optima, limiting further improvement within the fixed 100-iteration budget.

Compared to the strongest previously reported results, including the closed-source AlphaEvolve, **EvoX matches or exceeds AlphaEvolve on 5 out of 7 tasks**, such as Circle Packing, Circle Packing Rect, and MinMaxMinDist ($d = 3$), within 100 iterations. Since AlphaEvolve’s iteration budget is not publicly specified, direct cost comparison is not possible. We therefore treat AlphaEvolve’s numbers as published reference values rather than a controlled head-to-head comparison. These mathematical tasks are open construction problems with best-known values, not known optima. In particular, Circle Packing Rect is nearly saturated, with full-precision values of 2.36583237 for EvoX versus 2.36583213 for AlphaEvolve.

Table 1: **Main results for math optimization problems.** We report mean and best over three runs. “↑” / “↓” indicate maximization / minimization. All open methods use 100 iterations under the same backbone model. **Bold** denotes the best LLM-based open framework result per model. Green cells indicate matches or improvements over human SOTA or closed-source AlphaEvolve results. In Circle Packing Rect, EvoX achieves 2.36583237, exceeding AlphaEvolve’s 2.36583213, although both appear equal when rounded.

Task	Stat	Human	AlphaEvolve	GPT-5				Gemini-3.0-Pro			
				OpenEvolve	GEPA	Shinka	EvoX	OpenEvolve	GEPA	Shinka	EvoX
Circle Packing (↑)	Mean	–	–	2.5308	2.6129	2.4642	2.6324	2.5414	2.6198	2.6216	2.6328
	Best	2.6340	2.6350	2.5414	2.6285	2.5161	2.6359	2.5414	2.6208	2.6358	2.6359
Circle Packing Rect (↑)	Mean	–	–	2.2673	2.3265	2.3347	2.3525	2.3651	2.2160	2.3658	2.3658
	Best	2.3640	2.3658	2.2756	2.3537	2.3577	2.3600	2.3658	2.2507	2.3658	2.3658
heilbronn.convex (↑)	Mean	–	–	0.0230	0.0259	0.0228	0.0263	0.0279	0.0228	0.0279	0.0287
	Best	0.0306	0.0309	0.0267	0.0269	0.0256	0.0272	0.0280	0.0274	0.0287	0.0296
heilbronn.triangle (↑)	Mean	–	–	0.0250	0.0317	0.0319	0.0316	0.0331	0.0311	0.0351	0.0359
	Best	0.0360	0.0365	0.0283	0.0332	0.0338	0.0339	0.0350	0.0329	0.0356	0.0365
min_max_min_dist ($n=16, d=2$) (↓)	Mean	–	–	13.00	12.98	12.99	12.95	12.96	12.89	12.89	12.89
	Best	12.89	12.89	13.00	12.95	12.98	12.89	12.89	12.89	12.89	12.89
min_max_min_dist ($n=14, d=3$) (↓)	Mean	–	–	4.51	4.30	4.19	4.21	4.17	4.71	4.16	4.17
	Best	4.17	4.17	4.46	4.18	4.17	4.18	4.16	4.59	4.16	4.16
third_autocorr_ineq (↓)	Mean	–	–	1.4671	1.4768	1.4785	1.4714	1.4609	1.4678	1.4595	1.4589
	Best	1.4581	1.4557	1.4610	1.4758	1.4614	1.4609	1.4600	1.4598	1.4578	1.4558
signal processing (↑)	Mean	–	–	0.5686	0.6891	0.4855	0.7106	0.5525	0.6169	0.4799	0.7351
	Best	–	–	0.6219	0.7057	0.5328	0.7214	0.5649	0.6798	0.5049	0.7429

Beyond aggregate scores, EvoX discovers qualitatively distinct solutions: hexagonal-lattice + SLSQP refinement for circle packing (state-of-the-art), and a 14-point $\mathbb{R}^3$ configuration for MinMaxMinDist via constrained extremal-distance optimization from polyhedral seeds. These solutions outperform prior systems with manually designed search strategies. The discovered programs are listed in Appendix F.

5.2 Main Results: System Performance and Algorithm Problems

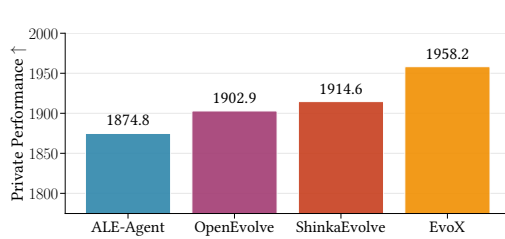
Systems Performance Problems. We also evaluate EvoX on six real-world system optimization tasks spanning expert placement load balancing (EPLB), GPU sharing (PRISM), LLM-driven analytics (LLM-SQL), multi-cloud broadcast optimization (Cloudcast), transaction scheduling, and telemetry repair (Table 2). EvoX **exceeds human-best results on all six benchmarks** under Gemini-3.0-Pro. For mean performance, EvoX achieves the best score on all tasks under GPT-5 and 5 of 6 tasks under Gemini-3.0-Pro. OpenEvolve achieves a slightly higher mean only for telemetry repair. For best performance, EvoX outperforms or ties all baselines on all tasks under GPT-5 and 5 of 6 tasks under Gemini-3.0-Pro.

EvoX uncovers new system optimization algorithms. In Cloudcast, EvoX discovers a Steiner-tree-based multicast routing strategy that reduces transfer cost beyond prior heuristics. In PRISM, EvoX identifies a model-placement strategy that minimizes peak KV-cache pressure by binary-searching a global load threshold and applying Best-Fit-Decreasing packing.

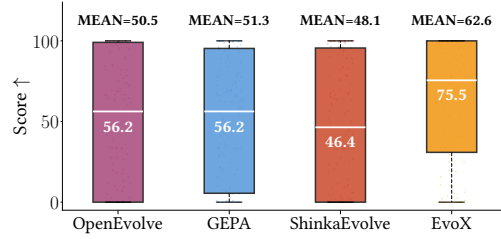
Algorithmic and Research Problems. As shown in Figure 2, we evaluate EvoX on ALE-Bench-Lite (Imajuku et al. (2025)) (10 tasks from AtCoder Heuristic Contests) and FrontierCS (Mang et al. (2025)) (172 open-ended programming tasks). On ALE-Bench-Lite, EvoX achieves the highest average private score (1958.2), outperforming ALE-Agent (1874.8), OpenEvolve (1902.9), and ShinkaEvolve (1914.6). On FrontierCS, EvoX reaches a mean of 62.6 and median of 75.5, improving over OpenEvolve by 24%/34%, GEPA by 22%/34%, and ShinkaEvolve by 30%/63%, demonstrating stronger solution quality and consistency.

Table 2: **Main results for system problems.** We report mean and best scores over three runs. “↑” denotes maximization and “↓” minimization. All methods (except human / prior AI) are run for 100 iterations using GPT-5 and Gemini-3.0-Pro. **Bold** denotes the best LLM-based open framework result per model. Green cells indicate matches or improvements over human SOTA results.

Task	Stat	Human	GPT-5				Gemini-3.0-Pro			
			OpenEvolve	GEPA	Shinka	EvoX	OpenEvolve	GEPA	Shinka	EvoX
EPLB (↑)	Mean	–	0.1300	0.1338	0.1185	0.1358	0.1272	0.1268	0.1206	0.1392
	Best	0.1265	0.1272	0.1445	0.1272	0.1453	0.1272	0.1272	0.1272	0.1453
PRISM (↑)	Mean	–	25.15	26.19	26.26	27.67	26.24	26.16	26.25	26.26
	Best	21.89	26.23	26.23	26.26	30.52	26.24	26.19	26.26	26.26
LLM-SQL (↑)	Mean	–	0.7053	0.7127	0.7123	0.7231	0.7251	0.7129	0.7210	0.7278
	Best	0.6920	0.7155	0.7129	0.7125	0.7298	0.7258	0.7134	0.7212	0.7300
Cloudcast (↓)	Mean	–	851.72	689.89	954.79	662.26	707.82	720.38	949.79	623.69
	Best	626.24	729.80	645.72	812.74	637.14	707.82	667.06	1032.42	623.69
Transaction Scheduling (↑)	Mean	–	3860.1	3752.6	4090	4292.32	4109.19	3615.57	3931.67	4267.75
	Best	2724.8	4237.3	3984.1	4329	4347.83	4273.50	3615.57	4255.32	4310.34
Telemetry Repair (↑)	Mean	–	0.9031	0.9158	0.9229	0.9464	0.9541	0.8505	0.9176	0.9381
	Best	0.8222	0.9515	0.9477	0.9515	0.9520	0.9541	0.8553	0.9331	0.9467



(a) Average performance across ALE-Bench-Lite.



(b) Performance on 172 FrontierCS tasks.

Figure 2: **Algorithm and Research Challenges.** EvoX achieves strongest performance across 10 different ALE-Bench-Lite problems and 172 different FrontierCS challenges.

5.3 Ablations

Starting from Different Search Strategies. Figure 3a shows EvoX’s behavior across different strategy initializations (Beam, Best-of- N , Top- K , MAP-Elites). Beam search and Best-of- N provide the strongest initializations by concentrating compute on repeatedly refining top candidates. Top- K instead preserves multiple candidates while MAP-Elites maintains a grid of diverse solutions and lags behind. Regardless of initialization, EvoX consistently improves by adapting its search strategy during optimization. We further study stronger initializations with adaptive policies from AdaEvolve (Cemri et al., 2026) in Appendix A.3.

Cost and Scaling. Figure 3b shows that EvoX and GEPA exceed 0.031 for under \$1, compared to \$7.6 for ShinkaEvolve and \$15.4 for OpenEvolve. GEPA plateaus at 0.0323 after 20 iterations, while EvoX continues improving to 0.0339 by adapting its search strategy. Overall, EvoX achieves comparable or better results at more than $\sim 3\times$ lower cost than Claude Code and other concurrent systems (see Appendix A.1, A.2).

Is informed strategy adaptation necessary? Table 3 isolates the effect of informed adaptation while holding both the underlying model (GPT-5) and the budget (100 total calls) fixed. The No-Cond control generates a new strategy at each stagnation check using GPT-5, but does not condition on either the strategy history $\mathcal{H}$ or the population descriptor ϕ . Strategy switching alone is beneficial: No-Cond improves over the static/random policy on all four tasks, including an increase in mean Signal Processing performance from 0.548 to 0.640.

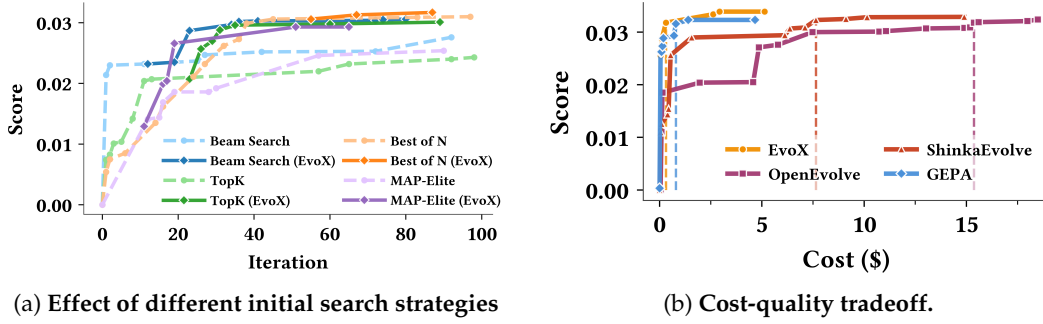


Figure 3: **Heilbronn triangle**. (a) EvoX (solid) initialized from different strategies continues improving beyond fixed baselines (dashed). (b) Cost-quality tradeoff (GPT-5): EvoX and GEPA reach > 0.031 for less than \$1, vs. ShinkaEvolve (\$7.6) and OpenEvolve (\$15.4).

Conditioning strategy generation on the current search state provides further gains: EvoX raises the Signal Processing mean to 0.711, achieves the best mean on three of four tasks, and obtains the best single-run result on all four. These results show that the gains do not come only from periodically switching strategies; using the search history and population state to guide those switches yields additional, task-dependent improvements.

Table 3: **Ablation of informed strategy adaptation**. All variants use GPT-5 under the same budget of 100 total calls. We report mean and best scores over three runs. Static/random uses a fixed random strategy-selection policy; No-Cond generates a new strategy at each stagnation check without access to the strategy history $\mathcal{H}$ or population descriptor ϕ ; EvoX conditions strategy generation on both. “ $\uparrow$ ” denotes maximization and “ $\downarrow$ ” minimization. **Bold** denotes the best result in each column.

Statistic	Variant	Circle Pack. ($\uparrow$)	Signal ($\uparrow$)	PRISM ($\uparrow$)	Cloudcast ($\downarrow$)
Mean	Static/random	2.4905	.5484	26.23	671
	No-Cond	2.5649	.6396	26.244	655
	EvoX	2.6324	.7106	27.67	674
Best	Static/random	2.5008	.5858	26.23	650
	No-Cond	2.6117	.6778	26.250	644
	EvoX	2.6359	.7214	30.52	642

Additional Results. Appendix A presents additional ablations of EvoX under different settings, including the results and limitations of using weaker open-source models such as Qwen3-8B for meta- and solution-evolution (Appendix A.5), and a variant with free co-evolved variation operators (Appendix A.4). Appendix B additionally reports equal-budget resource accounting (Appendix B.2), comparisons with fixed, annealed, and hand-designed adaptive baselines (Appendix B.3), extended analysis of the conditioning ablation (Appendix B.4), strategy-score and prompt-sensitivity ablations (Appendices B.6 and B.7), and an expanded four-task Qwen3-8B evaluation (Appendix B.8). Appendix C provides further case studies of search strategy evolution.

6 Conclusion

We introduced EvoX, a meta-learning method that jointly evolves candidate solutions and the search strategies that generate them. By adapting search strategy to the structure and stage of the optimization process, EvoX delivers consistent improvements in solution quality and cost efficiency across diverse domains. As optimization problems continue to scale in size and complexity, these results point toward a future of general-purpose, self-evolving systems that continuously refine how they search, reducing reliance on fixed, manually designed procedures.

Acknowledgments

This research is supported by NSF (IFML) CCF-2019844 and gifts from Accenture, AMD, Anyscale, Broadcom Inc., Google, IBM, Intel, Intesa Sanpaolo, Lambda, Mibura Inc., Samsung SDS, and SAP. We also thank Jiarong Xing, Asankhaya Sharma, and Joseph E. Gonzalez for their valuable feedback and discussion.

References

- Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziem, 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.
- Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W. Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando de Freitas. Learning to learn by gradient descent by gradient descent. In *Advances in Neural Information Processing Systems*, volume 29, pp. 3981–3989, 2016.
- Anthropic. Claude code documentation. <https://code.claude.com/docs/en/overview>, 2026. Accessed: 2026-03-31.
- 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.
- Zhicheng Cai, Xinyuan Guo, Yu Pei, Jiangtao Feng, Jinsong Su, Jiangjie Chen, Ya-Qin Zhang, Wei-Ying Ma, Mingxuan Wang, and Hao Zhou. Flex: Continuous agent evolution via forward learning from experience. *arXiv preprint arXiv:2511.06449*, 2025.
- Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026. doi: 10.48550/arXiv.2602.20133. URL <https://arxiv.org/abs/2602.20133>.
- Tianlong Chen, Xiaohan Chen, Wuyang Chen, Howard Heaton, Jialin Liu, Zhangyang Wang, and Wotao Yin. Learning to optimize: A primer and a benchmark. *Journal of Machine Learning Research*, 23(189):1–59, 2022.
- Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, et al. Symbolic discovery of optimization algorithms. *Advances in neural information processing systems*, 36:49205–49233, 2023.
- Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Shubham Agarwal, Mert Cemri, Bowen Wang, Alexander Krentsel, Tian Xia, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Ashwin Naren, Shulu Li, Ruiying Ma, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Let the barbarians in: How ai can accelerate systems performance research, 2025a. URL <https://arxiv.org/abs/2512.14806>.
- Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, et al. Barbarians at the gate: How AI is upending systems research. *arXiv preprint arXiv:2510.06189*, 2025b.
- Francois Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. Arcagi-2: A new challenge for frontier ai reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.
- Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025.

- Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A benchmark for long-horizon objective-driven algorithm engineering. *arXiv preprint arXiv:2506.09050*, 2025.
- Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.
- Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution, 2025. URL <https://arxiv.org/abs/2509.19349>.
- 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*, 2024a. URL <https://arxiv.org/abs/2401.02051>.
- Shengcai Liu, Caishun Chen, Xinghua Qu, Ke Tang, and Yew-Soon Ong. Large language models as evolutionary optimizers. In *2024 IEEE Congress on Evolutionary Computation*, pp. 1–8, 2024b. doi: 10.1109/CEC60901.2024.10611913. URL <https://arxiv.org/abs/2310.19046>.
- Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.
- Qiuyang Mang, Wenhao Chai, Zhifei Li, Huanzhi Mao, Shang Zhou, Alexander Du, Hanchen Li, Shu Liu, Edwin Chen, Yichuan Wang, et al. Frontiercs: Evolving challenges for evolving intelligence. *arXiv preprint arXiv:2512.15699*, 2025.
- Luke Metz, Niru Maheswaranathan, Jeremy Nixon, Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *International Conference on Machine Learning*, volume 97, pp. 4556–4565. PMLR, 2019.
- Alexander Novikov, Ngan Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.
- Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as operating systems, 2023. URL <https://arxiv.org/abs/2310.08560>.
- Yuxiao Qu, Matthew Y. R. Yang, Amrith Setlur, Lewis Tunstall, Edward Emanuel Beeching, Ruslan Salakhutdinov, and Aviral Kumar. Optimizing test-time compute via meta reinforcement fine-tuning, 2025. URL <https://arxiv.org/abs/2503.07572>.
- Bernardino Romera-Paredes, Mohammadamin Barekatin, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL <https://doi.org/10.1038/s41586-023-06924-6>.
- Asankhaya Sharma. OpenEvolve: an open-source evolutionary coding agent. GitHub, 2025. URL <https://github.com/codelion/openevolve>.
- Belle A Sheno. *Introduction to digital signal processing and filter design*, volume 169. John Wiley & Sons, 2005.
- Yiding Shi, Jianan Zhou, Wen Song, Jieyi Bi, Yaoxin Wu, and Jie Zhang. Generalizable heuristic generation through large language models with meta-optimization. In *International Conference on Learning Representations*, 2026. URL <https://arxiv.org/abs/2505.20881>.

- Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023. URL <https://arxiv.org/abs/2303.11366>.
- 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.
- Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, Chi Wang, et al. Pacevolve: Enabling long-horizon progress-aware consistent evolution. *arXiv preprint arXiv:2601.10657*, 2026.
- An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, et al. Qwen3 technical report, 2025. URL <https://arxiv.org/abs/2505.09388>.
- Mert Yuksekgonul, Daniel Kocaja, 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.
- Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024. URL <https://arxiv.org/abs/2310.02304>.
- Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents, 2026a. URL <https://arxiv.org/abs/2505.22954>.
- Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026b.
- Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

A Additional Results

We report additional evaluations to examine EvoX behavior beyond the primary benchmarks discussed in the main paper. Appendix A.1 compares EvoX against Claude Code, an agentic coding baseline, across all 12 benchmarks. Appendix A.2 evaluates against HyperAgents (Zhang et al. (2026b)), a concurrent work that has a similar self-improvement idea. Appendix A.3 studies initializing EvoX from AdaEvolve (Cemri et al., 2026). Appendix A.4 ablates the effect of freely exploring variation operators beyond the fixed set. Appendix A.5 evaluates EvoX with an open-source model (Qwen3-8B), demonstrating its generalizability and sustained improvements even under weaker generation quality. Appendix A.6 provides full per-task results against CodeEvolve, ThetaEvolve, and random sampling. Appendix A.7 reports results on the ARC-AGI reasoning benchmark.

A.1 EvoX Results vs. Claude Code

Setup. We compare EvoX against Claude Code (Anthropic (2026)), an agentic coding assistant that iteratively improves solutions through multi-turn LLM interactions. Both systems use the same backbone models (GPT-5 and Gemini-3-Pro) and identical benchmark evaluators; the only difference is the search method. Claude Code runs until convergence or budget exhaustion, while EvoX runs for 100 iterations. All costs are cumulative API spend in USD.

Results. Table 4 reports final scores and total costs across all 12 benchmarks. With GPT-5, EvoX matches or outperforms Claude Code on all 12 benchmarks (strictly better on 11, tied on 1) at $3.0\times$ **lower average cost** (\$6.4 vs. \$19.1). With Gemini-3-Pro, EvoX again matches or outperforms on all 12 (strictly better on 7, tied on 5) at $1.9\times$ **lower cost** (\$8.4 vs. \$16.0). The largest quality gaps appear on Signal Processing (0.72 vs. 0.53, GPT-5), Heilbronn Convex (0.027 vs. 0.019), and Cloudcast (637 vs. 704).

Discussion. Claude Code operates with full agentic flexibility: it can read and write files, execute code, maintain state across turns, and freely explore different solution paths, with the only constraint being a fixed evaluator. This flexibility enables rich reasoning and iterative refinement, but much of the computation is spent on local exploration, e.g., extended multi-turn reasoning, solution consolidation, and incremental improvements.

However, this process is largely implicit: the system does not explicitly structure how past solutions are reused or how exploration should adapt based on accumulated progress. As a result, it often revisits similar regions of the search space and requires more iterations to identify effective search patterns.

Figure 4 shows score-vs-cost trajectories on six benchmarks: EvoX (stars) consistently reaches equal or higher scores at $2\text{--}5\times$ lower cost than Claude Code (curves).

In contrast, EvoX introduces an explicit structure over solutions and strategies, where the strategies being evolved often include population management, score-aware parent selection, and problem-specific features to exploit the solution space. This allows the system to systematically leverage prior progress and experiences over time in a much more cost-efficient way.

A.2 EvoX Results vs. HyperAgents

Setup. HyperAgents (DGM-H) (Zhang et al., 2026b) extends the Darwin Gödel Machine (Zhang et al., 2026a) with self-referential self-improvement, combining a task agent and meta agent into a single editable program. Its search structure (parent selection, archive management) is inherited from the DGM and largely remains fixed throughout the run (Section 3 of (Zhang et al., 2026b)). EvoX instead treats the search strategy as an evolvable object, adapting parent selection, variation operators, and context construction based on progress. We compare both systems on 4 benchmarks (GPT-5) with a \$50 budget for HyperAgents and 100 iterations (\$5.9–\$6.9) for EvoX.

Table 4: EvoX vs. Claude Code results across 12 benchmarks. **Bold** = better or tied score per model. $\uparrow/\downarrow$: higher/lower is better. Cost = total API spend (USD).

Benchmark	GPT-5				Gemini-3-Pro			
	CC	EvoX	CC \$	EvoX \$	CC	EvoX	CC \$	EvoX \$
<i>Math</i>								
Circle Packing ($\uparrow$)	2.483	2.636	31.3	6.7	2.629	2.636	20.2	8.6
Circle Pack. Rect ($\uparrow$)	2.281	2.360	20.9	6.3	2.366	2.366	14.9	8.4
Min MaxMin $d=2$ ($\downarrow$)	13.00	12.89	15.2	6.2	12.89	12.89	12.1	8.1
Min MaxMin $d=3$ ($\downarrow$)	4.322	4.180	19.3	6.5	4.166	4.160	13.1	8.6
Heilbronn Tri. ($\uparrow$)	0.032	0.034	15.9	6.3	0.037	0.037	19.5	8.2
Heilbronn Cvx. ($\uparrow$)	0.019	0.027	22.8	6.3	0.027	0.030	15.5	8.2
Signal Proc. ($\uparrow$)	0.527	0.721	12.1	6.8	0.720	0.743	15.8	9.1
<i>Systems</i>								
PRISM ($\uparrow$)	25.61	30.52	16.4	5.9	26.26	26.26	12.1	7.4
Cloudcast ($\downarrow$)	703.7	637.1	29.2	6.9	669.2	623.7	23.6	9.3
LLM-SQL ($\uparrow$)	0.729	0.730	13.3	6.3	0.730	0.730	12.6	8.4
Transaction Scheduling ($\uparrow$)	4016	4348	14.6	6.5	4237	4310	21.8	8.7
EPLB ($\uparrow$)	0.145	0.145	18.2	6.3	0.145	0.145	10.9	8.1
Average cost	—	—	19.1	6.4	—	—	16.0	8.4

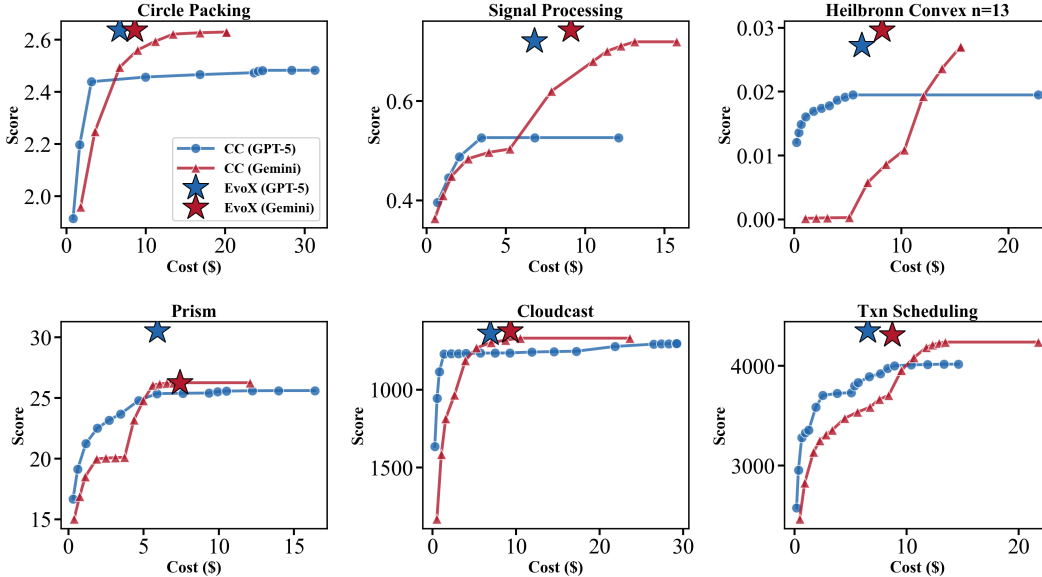


Figure 4: Score vs. cumulative API cost for EvoX (stars) and Claude Code (curves) on six benchmarks. EvoX reaches higher scores at a fraction of the cost. For Cloudcast, lower is better (y-axis inverted).

Results. Table 5 and Figure 5 compare scores at the \$50 budget for HyperAgents. Within this budget, EvoX outperforms HyperAgents on all 4 benchmarks while spending only \$5.9–\$6.9. HyperAgents does not reach EvoX’s score on any benchmark within \$50: it reaches 2.540 vs. 2.636 on Circle Packing, 0.613 vs. 0.721 on Signal Processing, 26.23 vs. 30.52 on PRISM, and 657.2 vs. 637.1 on Cloudcast.

Discussion. The key structural difference is how the two systems learn from experience. HyperAgents inherits a fixed evolutionary loop from the DGM: parent selection and the archive structure are handcrafted and do not change during the run. EvoX, in contrast, explicitly separates the search strategy as a first-class evolvable object, enabling it to adapt how parents are selected, which variation operators are applied, and how past solutions inform future generations. This structured approach to strategy evolution allows EvoX to extract more value from each iteration, achieving stronger scores at a fraction of the cost on these benchmarks.

Table 5: EvoX vs. HyperAgents at \$50 budget (GPT-5). **Bold** = better score. $\uparrow/\downarrow$: higher/lower is better.

Benchmark	EvoX Score	EvoX \$	HA Score @\$50
Circle Packing ($\uparrow$)	2.636	6.7	2.540
Signal Proc. ($\uparrow$)	0.721	6.8	0.613
PRISM ($\uparrow$)	30.52	5.9	26.23
Cloudcast ($\downarrow$)	637.1	6.9	657.2

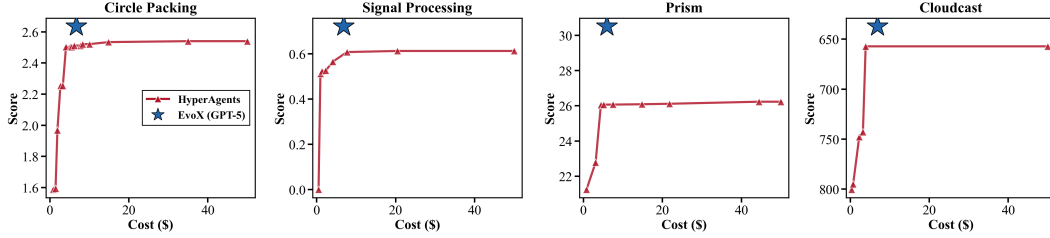


Figure 5: Score vs. cumulative API cost (capped at \$50) for HyperAgents (red) and EvoX (blue star) on 4 benchmarks (GPT-5). EvoX reaches its final score at \$5.9–\$6.9. HyperAgents does not reach EvoX’s score on any benchmark within the \$50 budget. For Cloudcast, lower is better (y-axis inverted).

A.3 Ablation: Evolving from AdaEvolve

Setup. AdaEvolve (Cemri et al., 2026) applies adaptive parameter control (e.g., per-iteration intensity tuning, UCB-based island selection, and meta-guided tactic generation) within a fixed search architecture. EvoX can use any search strategy as its starting strategy and evolve it further. We run EvoX initialized from AdaEvolve’s search strategy for 100 iterations on 4 shared benchmarks (GPT-5) and compare against AdaEvolve alone and EvoX initialized with a random sampling policy, each also at 100 iterations.

Results. Table 6 and Figure 6 report final scores. Individual AdaEvolve and EvoX (initialized with random sampling policy) scores are the best of 3 runs from the respective main results; the combined configuration is a single run. EvoX initialized from AdaEvolve outperforms both individual systems on 3 of 4 benchmarks: Signal Processing (0.760 vs. next-best 0.721), Heilbronn Convex (0.0291 vs. 0.029), and Cloudcast (623.4 vs. 637.1). On PRISM, EvoX initialized from random sampling achieves 30.52 in its best run (mean 27.67), which the AdaEvolve-initialized variant (26.27) does not match. This suggests that starting from a simpler policy can enable the discovery of qualitatively different strategies that are not reachable when initialized from AdaEvolve.

Table 6: EvoX initialized from AdaEvolve vs. each system alone (100 iterations, GPT-5). **Bold** = best score. $\uparrow/\downarrow$: higher/lower is better.

Benchmark	AdaEvolve	EvoX (Random Sampling)	EvoX (AdaEvolve)
Signal Proc. ($\uparrow$)	0.718	0.721	0.760
Heilbronn Cvx. ($\uparrow$)	0.029	0.027	0.0291
Cloudcast ($\downarrow$)	640.5	637.1	623.4
PRISM ($\uparrow$)	26.37	30.52	26.27

Discussion. On 3 of 4 benchmarks, starting from AdaEvolve gives EvoX the best of both worlds: fast early convergence from AdaEvolve’s per-iteration adaptation, followed by strategy-level improvements that push beyond either system alone. On PRISM, however, starting from scratch lets EvoX explore a wider strategy space and discover a high-scoring outlier (30.52) that the AdaEvolve-initialized run does not reach; a structured starting point can constrain exploration on tasks where the optimal strategy diverges significantly from the

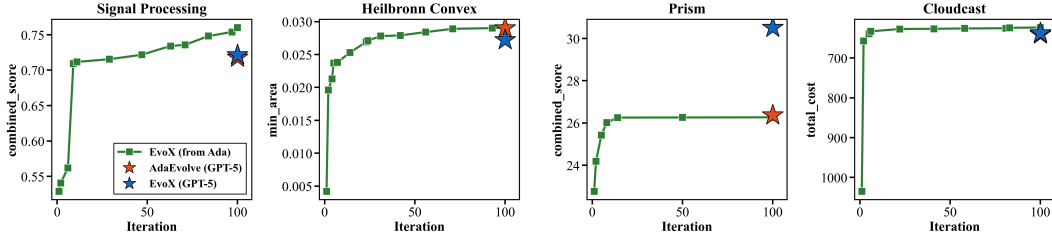


Figure 6: Best-so-far score vs. iteration for EvoX initialized from AdaEvolve (green curve) on 4 benchmarks (100 iterations, GPT-5). Stars mark the scores of AdaEvolve (red) and EvoX (blue) run individually. The combined run exceeds both on 3 of 4 benchmarks; on PRISM, EvoX alone achieves a higher score. For Cloudcast, lower is better.

initial one. These results suggest the two approaches are complementary: AdaEvolve-style initialization helps when the starting algorithm is close to optimal, while scratch-start EvoX retains more freedom to discover qualitatively different strategies.

A.4 Ablation: Freely Exploring Variation Operators

Setup. In standard EvoX, the search strategy selects among three simple variation operators (Section 3) throughout the run. In this ablation, we remove this constraint: the LLM freely generates and updates variation prompts alongside the search strategy, with no fixed operator structure imposed. We run this configuration for 500 iterations on 4 benchmarks (GPT-5).

Results. Table 7 and Figure 7 report the results. With free operator exploration, EvoX surpasses the three-operator configuration on 3 of 4 benchmarks by iteration 500: Heilbronn Convex (+16.2%), Signal Processing (+3.3%), and Cloudcast (+2.5%). However, reaching the three-operator score obtained under the 100-iteration budget takes substantially longer: 442 iterations on Signal Processing, 173 on Heilbronn Convex, and 47 on Cloudcast (Table 7, rightmost column). On PRISM, allowing the model to define operators freely does not match the three-operator result (27.55 vs. 30.52) at the end of 500 iterations. The slower convergence reflects the cost of discovering effective operators from scratch rather than using a simple, predefined fixed set. Further ablation is needed to determine whether relaxing operator constraints yields consistent benefits across a broader range of tasks and longer horizons.

Table 7: Three fixed operators (100 iters) vs. freely explored operators (500 iters), GPT-5. **Bold** = better score. The rightmost column shows at which iteration the free-operator run first matches the three-operator score. $\uparrow/\downarrow$: higher/lower is better.

Benchmark	Three Ops @100	Free Ops @500	Iter to match
Signal Proc. ($\uparrow$)	0.721	0.745	442
Heilbronn Cvx. ($\uparrow$)	0.027	0.032	173
PRISM ($\uparrow$)	30.52	27.55	never
Cloudcast ($\downarrow$)	637.1	621.2	47

Discussion. Freely exploring operators alongside strategies expands the search space: the system can discover task-specific operator types that the fixed refinement/structural-variation pair does not express. On Heilbronn Convex, this yields the largest gain (+16.2%), suggesting the optimal variation pattern for this non-convex landscape differs from the fixed operator set. The cost is slower convergence: without structured operator labels, the system must spend early iterations identifying effective operator types before making solution-level progress. On PRISM, this overhead is not recovered within 500 iterations, consistent with the observation that PRISM’s reward landscape already favors the standard operators. These results indicate that free operator exploration is a viable extension when the iteration budget permits, but fixed operators remain more sample-efficient for the 100-iteration regime.

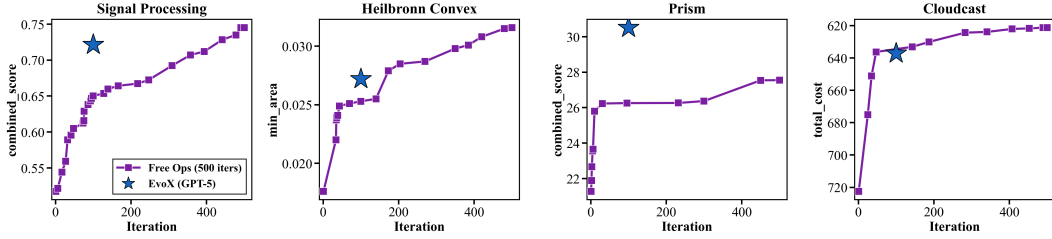


Figure 7: Best-so-far score vs. iteration for EvoX with freely explored operators (purple curve, 500 iterations) on 4 benchmarks (GPT-5). Blue stars mark the three-operator EvoX score at 100 iterations. The free-operator run surpasses the three-operator result on 3 of 4 benchmarks but requires substantially more iterations. For Cloudcast, lower is better.

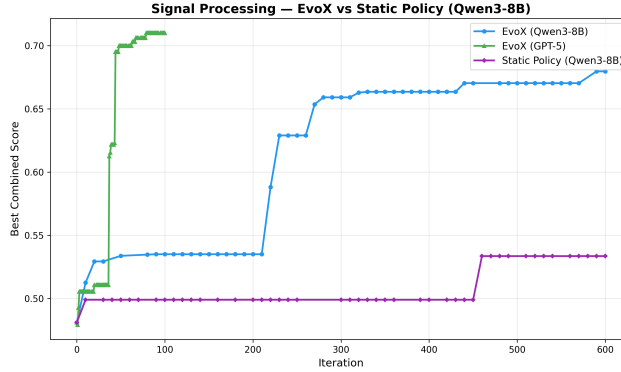


Figure 8: Best-so-far score vs. iteration on Signal Processing comparing EvoX with Qwen3-8B (blue), EvoX with GPT-5 (green), and a static-policy baseline (purple). EvoX with Qwen3-8B shows slower, step-wise improvements with extended plateaus, reflecting delayed discovery of effective search strategies under weaker models. In contrast, GPT-5 reaches similar improvements much earlier.

A.5 Ablation: Open-Source Model Results (Qwen3-8B)

Setup. We also run EvoX with open-source model, using Qwen3-8B (Yang et al., 2025) as the backbone model for both solution and strategy generation on the Signal Processing benchmark, using the same setup as in the main experiments. We compare EvoX (with strategy evolution) against a static-policy baseline and additionally analyze longer runs to study convergence under weaker models.

Results. Figure 8 shows that under Qwen3-8B, EvoX exhibits slower and noisier early progress due to lower generation quality, but continues to improve over time and significantly outperforms the static-policy baseline of random sampling policy. Compared to GPT-5, it requires more iterations to reach similar performance, indicating that weaker models benefit from longer optimization horizons; nevertheless, the gains over static baselines remain substantial.

Discussion. These results suggest that meta evolution on the search or optimization strategy acts as a *learning-from-experience* mechanism: although weaker models produce noisier generations, EvoX accumulates feedback across strategy deployments to gradually identify effective patterns and improve over time. This enables it to compensate for lower generation quality through longer-horizon adaptation. Unlike static policies that plateau early, EvoX continues improving by refining both *what* and *how* to generate, indicating that its benefits extend to weaker models given sufficient iterations.

Table 8: Main results for math optimization problems. We report mean and best over three runs. “↑” / “↓” indicate maximization / minimization.

Task	Stat	Human	AlphaEvolve	ThetaEvolve	CodeEvolve	GPT-5					Gemini-3.0-Pro				
						Random	OpenEvolve	GEPA	Shinka	EvoX	Random	OpenEvolve	GEPA	Shinka	EvoX
Circle Packing (↑)	Mean	–	–	–	–	2.4905	2.5308	2.6129	2.4642	2.6324	2.5564	2.5414	2.6198	2.6216	2.6328
	Best	2.6340	2.6350	2.6360	2.6360	2.5000	2.5414	2.6285	2.5078	2.6360	2.5853	2.5414	2.6208	2.6358	2.6360
Circle Packing Rect (↑)	Mean	–	–	–	–	2.2105	2.2673	2.3265	2.3347	2.3525	2.3432	2.3651	2.2160	2.3658	2.3658
	Best	2.3640	2.3658	–	2.3658	2.2783	2.2756	2.3537	2.3577	2.3600	2.3652	2.3658	2.2507	2.3658	2.3658
heilbronn.convex (↑)	Mean	–	–	–	–	0.0185	0.0230	0.0259	0.0228	0.0263	0.0285	0.0279	0.0228	0.0279	0.0287
	Best	0.0306	0.0309	–	–	0.0198	0.0267	0.0269	0.0256	0.0272	0.0298	0.0280	0.0274	0.0287	0.0296
heilbronn.triangle (↑)	Mean	–	–	–	–	0.0261	0.0250	0.0317	0.0319	0.0316	0.0318	0.0331	0.0311	0.0351	0.0359
	Best	0.0360	0.0365	–	–	0.0285	0.0283	0.0332	0.0338	0.0339	0.0329	0.0350	0.0329	0.0356	0.0365
min_max_min_dist ($n=16, d=2$) (↓)	Mean	–	–	–	–	13.00	13.00	12.98	12.99	12.95	12.92	12.96	12.89	12.89	12.89
	Best	12.89	12.89	–	12.89	13.00	13.00	12.95	12.98	12.89	12.89	12.89	12.89	12.89	12.89
min_max_min_dist ($n=14, d=3$) (↓)	Mean	–	–	–	–	4.47	4.51	4.19	4.19	4.21	4.19	4.17	4.71	4.16	4.17
	Best	4.17	4.17	–	4.17	4.46	4.46	4.17	4.17	4.18	4.16	4.16	4.59	4.16	4.16
third_autocorr_ineq (↓)	Mean	–	–	–	–	1.4820	1.4671	1.4768	1.4785	1.4714	1.4594	1.4609	1.4678	1.4595	1.4589
	Best	1.4581	1.4557	1.4930	–	1.4607	1.4610	1.4758	1.4614	1.4609	1.4561	1.4600	1.4598	1.4578	1.4558
signal processing (↑)	Mean	–	–	–	–	0.5484	0.5686	0.6891	0.4855	0.6580	0.5527	0.5525	0.6169	0.4799	0.7351
	Best	–	–	–	–	0.5858	0.6219	0.7057	0.5328	0.6898	0.5647	0.5649	0.6798	0.5049	0.7429

Table 9: Main results for system problems. We report mean and best scores over three runs. “↑” denotes maximization and “↓” minimization.

Task	Stat	Human	GPT-5					Gemini-3.0-Pro				
			Random	OpenEvolve	GEPA	Shinka	EvoX	Random	OpenEvolve	GEPA	Shinka	EvoX
EPLB (↑)	Mean	–	0.1338	0.1300	0.1338	0.1185	0.1358	0.1280	0.1272	0.1268	0.1206	0.1392
	Best	0.1265	0.1445	0.1272	0.1445	0.1272	0.1453	0.1285	0.1272	0.1272	0.1272	0.1453
PRISM (↑)	Mean	–	26.23	25.15	26.19	26.26	27.67	26.23	26.24	26.16	26.25	26.26
	Best	21.89	26.23	26.23	26.23	26.26	30.52	26.23	26.24	26.19	26.26	26.26
LLM-SQL (↑)	Mean	–	0.7201	0.7053	0.7127	0.7123	0.7231	0.7380	0.7251	0.7129	0.7210	0.7278
	Best	0.6920	0.7289	0.7155	0.7129	0.7125	0.7298	0.7440	0.7258	0.7134	0.7212	0.7300
Cloudcast (↓)	Mean	–	659.00	851.72	689.89	954.79	662.26	629.84	707.82	720.38	949.79	623.69
	Best	626.24	644.84	729.80	645.72	812.74	637.14	635.72	707.82	667.06	1032.42	623.69
Transaction Scheduling (↑)	Mean	–	4123.87	3860.10	3752.6	4090.0	4292.32	4037.14	4109.19	3615.57	3931.67	4267.75
	Best	2724.8	4149.38	4237.3	3984.1	4329.0	4347.83	4237.29	4273.50	3615.57	4255.32	4310.34
Telemetry Repair (↑)	Mean	–	0.8959	0.9031	0.9158	0.9229	0.9464	0.9179	0.9541	0.8505	0.9176	0.9381
	Best	0.8222	0.9498	0.9515	0.9477	0.9515	0.9520	0.9515	0.9541	0.8553	0.9331	0.9467

A.6 EvoX v.s. Random Sampling Policy, ThetaEvolve, and CodeEvolve

We additionally compare against CodeEvolve (Assumpção et al. (2025)) and ThetaEvolve (Wang et al. (2025)), two recent systems representing strong prior approaches for LLM-driven optimization. We also detailed the performance of running random sampling policy naively without strategy evolution. Across tasks, EvoX maintains consistent performance advantages over both fixed-policy and adaptive-policy baselines. Detailed per-task breakdowns are provided below in Table 8 and Table 9.

A.7 Additional Benchmark: ARC-AGI-2

ARC-AGI-2 Tasks (Chollet et al. (2025)) evaluate abstract and compositional reasoning across programmatic problem-solving instances. Although ARC-AGI-2 is not explicitly designed as an optimization benchmark, it provides a useful testbed for analyzing cross-domain robustness.

Experiments follow the evaluation protocol described in Section 4.1. OpenEvolve (OE) and EvoX operate under a matched inference budget (30 LLM iterations) per task.

These results indicate that EvoX maintains performance improvements even on reasoning-oriented tasks outside traditional optimization settings. We emphasize that ARC-AGI fundamentally differs from the evolutionary optimization regime, as standard ARC evalua-

Table 10: EvoX performance on ARC-AGI-2 benchmarks. Values denote final accuracy.

Backbone	OpenEvolve	EvoX
GPT-5	41%	48%
Gemini-3-Pro	45%	51%

tion assumes strict train–test separation, whereas evolutionary frameworks typically adapt during inference.

Table 11: Comparison with related optimizer- and heuristic-evolution methods using GPT-5 and 100 total LLM calls. Entries are mean $\pm$ standard deviation | best over three runs.

Benchmark	Method	Mean $\pm$ std.	Best
Circle Packing ($\uparrow$)	STOP	2.470 $\pm$.06	2.510
	MoH	2.573 $\pm$.10	2.629
	EoH	2.541 $\pm$.02	2.561
	LMEA	2.513 $\pm$.03	2.548
	EvoX	2.6324$\pm$.003	2.6359
Signal Processing ($\uparrow$)	STOP	.558 $\pm$.04	.605
	MoH	.656 $\pm$.04	.701
	EoH	.535 $\pm$.03	.571
	LMEA	.533 $\pm$.02	.554
	EvoX	.7106$\pm$.010	.7214
PRISM ($\uparrow$)	STOP	26.44 $\pm$.32	26.80
	MoH	26.25 $\pm$.01	26.26
	EoH	26.26 $\pm$.01	26.27
	LMEA	26.46 $\pm$.05	26.52
	EvoX	27.67$\pm$2.02	30.52
Cloudcast ($\downarrow$)	STOP	1029 $\pm$ 11	1016
	MoH	696.3 $\pm$ 52.4	657.1
	EoH	676.7 $\pm$ 20	645
	LMEA	679.8 $\pm$ 24	646
	EvoX	674$\pm$21	642

B Additional Controlled Analyses

This section reports the additional controlled experiments and analyses conducted during the review period. Unless stated otherwise, results use GPT-5, three runs, and a fixed budget of 100 *total* LLM calls. For EvoX, this budget includes both solution-generation calls and all strategy-generation calls and retries.

B.1 Positioning Relative to Meta-Optimization and Heuristic Evolution

Several prior systems optimize an optimizer or use an LLM inside an evolutionary loop, but they differ in what is adapted during the target run. STOP learns an optimizer scaffold through a separate self-improvement procedure, and MoH meta-optimizes a heuristic optimizer across training tasks (Zelikman et al., 2024; Shi et al., 2026). In contrast, EvoX adapts its search-policy program online within each target run and conditions updates on the observed population and strategy history. EoH evolves heuristic ideas and implementations under a predefined evolutionary procedure, while LMEA uses a fixed population loop with an adaptive generation temperature (Liu et al., 2024a;b). Thus, the contribution here is not the first use of meta-optimization; it is the setting and mechanism of state-conditioned, online search-procedure adaptation under the same target-run budget used for solution search.

Table 11 gives a direct equal-backbone, equal-call comparison. It is an empirical target-run comparison rather than a claim that the original systems used identical training or deployment settings.

B.2 Equal-Budget Resource Accounting

All strategy calls and retries are charged to the same 100-call budget. Consequently, EvoX averages 93.2 evaluated solution candidates and 6.8 strategy-generation calls, whereas the fixed-loop baselines use all 100 calls for candidate generation. Table 12 reports benchmark-averaged resource use. EvoX is cost-matched with OpenEvolve (\$6.10 versus \$6.12), 24.7% cheaper than GEPA, and 22.8% cheaper than ShinkaEvolve. Its mean wall-clock time is 26.9% lower than GEPA’s.

Table 12: Average resource use across benchmarks under a fixed 100-total-LLM-call budget. Token counts include all calls.

Algorithm	LLM	Eval.	Input tok.	Output tok.	Algorithm	Cost	vs. EvoX	Time
EvoX	100	93.2	1,678,400	343,000	EvoX	\$6.10	1.00×	3.8h
OpenEvolve	100	100	1,579,100	370,000	OpenEvolve	\$6.12	1.00×	4.0h
GEPA	100	100	630,000	855,000	GEPA	\$8.10	1.33×	5.2h
ShinkaEvolve	100	100	850,000	775,000	ShinkaEvolve	\$7.90	1.29×	3.9h

(a) Calls and token volume

(b) Cost and wall time

B.3 Fixed, Scheduled, and Hand-Designed Adaptive Baselines

Table 13 compares EvoX with fixed or annealed OpenEvolve settings, fixed GEPA selection settings, and hand-designed adaptive methods. No single simpler configuration matches EvoX across the four tasks. These results do not establish that an LLM rewrite is the only possible adaptation mechanism. They show that the tested scalar schedules and hand-designed adaptive rules do not match a system that can rewrite selection logic, state, and credit assignment within the same budget.

Table 13: Fixed, annealed, and adaptive baselines under the same 100-call budget. Entries are mean $\pm$ standard deviation | best.

Configuration	Circle Packing ($\uparrow$)		Signal Proc. ($\uparrow$)		PRISM ($\uparrow$)		Cloudcast ($\downarrow$)	
OE ($e=.05$)	2.541 $\pm$.004	2.543	.526 $\pm$.045	.571	26.23 $\pm$.00	26.23	728 $\pm$ 45	679
OE ($e=.40$)	2.540 $\pm$.006	2.544	.651 $\pm$.038	.691	26.23 $\pm$.00	26.23	762 $\pm$ 52	710
OE ($e=.60$)	2.538 $\pm$.008	2.543	.613 $\pm$.042	.659	26.23 $\pm$.00	26.23	843 $\pm$ 58	771
OE (anneal .60 $\rightarrow$.05)	2.541 $\pm$.007	2.546	.571 $\pm$.048	.626	26.23 $\pm$.00	26.23	745 $\pm$ 48	699
GEPA (greedy)	2.483 $\pm$.018	2.497	.665 $\pm$.016	.685	26.23 $\pm$.00	26.23	672 $\pm$ 38	653
GEPA ($\epsilon=.30$)	2.541 $\pm$.008	2.547	.550 $\pm$.019	.573	26.23 $\pm$.00	26.23	678 $\pm$ 42	653
GEPA (Pareto)	2.541 $\pm$.006	2.545	.533 $\pm$.017	.552	26.23 $\pm$.00	26.23	715 $\pm$ 50	683
AdaEvolve	2.629 $\pm$.001	2.636	.698 $\pm$.017	.718	26.30 $\pm$.07	26.37	662.3$\pm$.1	640.5
ShinkaEvolve	2.464 $\pm$.083	2.541	.486 $\pm$.044	.533	26.26 $\pm$.00	26.26	954.8 $\pm$ 125	812.7
EvoX	2.632$\pm$.003	2.6359	.711$\pm$.010	.7214	27.67$\pm$2.02	30.52	674 $\pm$ 21	642

B.4 Conditioning Versus Unconditioned Strategy Switching

The No-Cond control generates a new strategy with the same LLM whenever stagnation is detected, but omits both the strategy history $\mathcal{H}$ and population descriptor ϕ . It therefore isolates strategy switching from informed, state-conditioned adaptation. Table 3 (Section 5.3) shows that unconditioned switching improves over the static random policy on all four tasks. Full EvoX then obtains the best mean on three of four tasks and the best single run on all four. No-Cond has the lower Cloudcast mean (655 versus 674), while EvoX has the lower best-run cost (642 versus 644). Thus, switching itself helps, and conditioning provides a further gain on most tasks rather than explaining performance through occasional policy replacement alone.

B.5 What the Strategy Generator Changes

A strategy update is an LLM-driven rewrite of the search-policy program, rather than a classical mutation of a fixed-length genome. Across 573 strategy changes from 14 benchmarks, two backbone models (GPT-5 and Gemini-3.0-Pro), and three runs per setting, the overlapping categories in Table 14 show that most updates change more than scalar parameters. The categories were assigned by Claude Code with Opus 4.8; because one update may have several labels, counts do not sum to 573.

Table 14: Overlapping categories of learned strategy changes.

Category	Share	Count	Examples of changes
Selection logic	63%	359	Parent choice, context construction, multi-objective pairing, diversity-aware sampling
Parameters	47%	271	Thresholds, weights, cooldowns, and exploration ratios
Structural additions	37%	214	Per-parent efficacy, lineage trees, usage counters, and approach tags
Algorithmic mechanisms	19%	106	UCB credit, technique categorization, and periodic statistic decay

The resulting programs average 329 lines of code (range 174–765), compared with 99 lines for the initial random policy. Candidate strategies pass API and functional checks 99.5% of the time. A rejected rewrite is not deployed: the current strategy remains active and generation is retried at the next stagnation check, one window (10% of iterations) later. This validity gate separates executable-code quality from design quality: broken programs are filtered, while valid strategies compete according to downstream score J . As noted above, strategy generation uses 6.8% of the 100-call budget on average.

Representative programs combine multiple mechanisms. In Signal Processing, the iteration-40 strategy selects high-smoothness parents, uses high-tracking-accuracy programs as context, and applies UCB to per-parent improvement history; the resulting candidates include hybrids of SSA and Whittaker smoothing, coinciding with the largest observed score jump (approximately +0.12). In Circle Packing, an evolved strategy detects near-duplicates, tags algorithmic families such as hexagonal lattices, concentric rings, and SLSQP-based optimization, and favors under-tried families. In PRISM, evolved state tracks per-parent trials, success rates, and best-child scores, then triggers structural exploration from lower-scoring but structurally diverse programs when no parent remains productive. Full per-window traces appear in Appendix C.

B.6 Strategy-Score Ablation

The strategy score is $J = \Delta \log(1 + s_{\text{start}}) / \sqrt{W}$, after converting each task to a higher-is-better utility. Here $\Delta = s_{\text{end}} - s_{\text{start}}$. The $\sqrt{W}$ factor prevents longer windows from being rewarded solely because they accumulate more opportunities for improvement, while the log factor gives additional weight to gains made near a higher incumbent frontier. Table 15 drops each component. The full score gives the best mean on three of four tasks; removing the log factor is best on Signal Processing, indicating that frontier weighting is regime-dependent rather than universally beneficial.

Table 15: Strategy-score ablation (mean $\pm$ standard deviation; GPT-5, 100 calls, three runs).

Variant	Circle Pack. ($\uparrow$)	Signal ($\uparrow$)	PRISM ($\uparrow$)	Cloudcast ($\downarrow$)
Full J	2.632$\pm$.003	.711 $\pm$.010	27.67$\pm$2.02	674$\pm$21
Raw (Δ)	2.575 $\pm$.082	.684 $\pm$.096	27.52 $\pm$ 2.10	677 $\pm$ 22
No log ($\Delta / \sqrt{W}$)	2.593 $\pm$.027	.738$\pm$.001	27.43 $\pm$ 2.15	681 $\pm$ 22
No $\sqrt{W}$ ($\Delta \log$)	2.571 $\pm$.052	.677 $\pm$.044	27.31 $\pm$ 2.20	683 $\pm$ 22

B.7 Prompt Sensitivity

We compare the default prompts with a paraphrase, one-line operator prompts, and a single generic modification instruction. Paraphrasing changes results by at most a few percent, while removing operator semantics degrades Signal Processing and Cloudcast by roughly 5–11%. The result suggests limited sensitivity to exact wording but meaningful dependence on conveying the explore/refine operator distinction. Table 16 reports the full results.

Table 16: Prompt sensitivity (mean $\pm$ standard deviation; GPT-5, 100 calls, three runs).

Variant	Circle Pack. ($\uparrow$)	Signal ($\uparrow$)	PRISM ($\uparrow$)	Cloudcast ($\downarrow$)
Default	2.632 $\pm$.003	.7106 $\pm$.010	27.67 $\pm$ 2.02	674 $\pm$ 21
Paraphrased	2.626 $\pm$.005	.704 $\pm$.018	27.38 $\pm$ 2.1	671 $\pm$ 30
Minimal operators	2.594 $\pm$.012	.647 $\pm$.025	26.52 $\pm$ 2.6	708 $\pm$ 28
No operator distinction	2.597 $\pm$.010	.636 $\pm$.034	26.71 $\pm$ 2.3	724 $\pm$ 41

B.8 Expanded Open-Source Model Evaluation

We extend the Qwen3-8B bf16, thinking-mode experiment to two math and two systems tasks, using six runs per cell. EvoX improves both mean and best over the static policy on all four tasks. A one-sided sign test over the four task-level mean comparisons gives $p = .0625$, the minimum attainable value for four paired task outcomes. A complementary run-level sign test over the 19 non-tied paired runs gives $p = .002$. Table 17 reports the per-task results.

Table 17: Qwen3-8B results with six runs per cell. Entries are mean $\pm$ standard deviation | best.

Task	Static policy		EvoX	
Circle Packing ($\uparrow$)	1.816 $\pm$.077	1.956	1.964$\pm$.331	2.626
Signal Processing ($\uparrow$)	.503 $\pm$.009	.522	.576$\pm$.073	.688
PRISM ($\uparrow$)	23.62 $\pm$.50	24.02	24.66$\pm$.84	25.91
Cloudcast ($\downarrow$)	659.16 $\pm$ 4.47	657.16	658.16$\pm$5.39	651.15

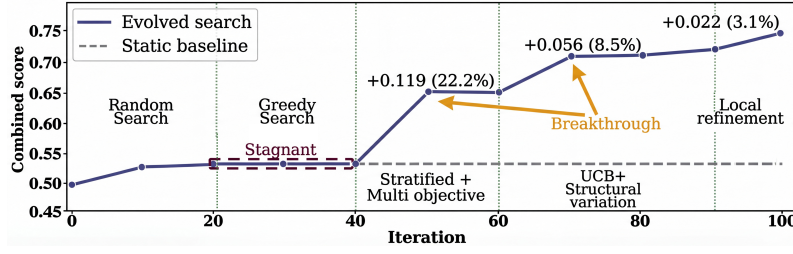


Figure 9: **Evolving search strategy on the signal processing task.** Starting from a uniform random sampling strategy, EvoX detects stagnation and adaptively switches to greedy search, stratified multi-objective sampling, UCB-guided structural variation, and finally local refinement. These strategy changes enable discovery of improved filtering programs and yield major gains at iterations ~ 48 (+0.119), ~ 70 (+0.056), and ~ 96 (+0.022), achieving 34.1% higher final score than the static baseline.

C Case Studies on Meta Evolution

C.1 Case Study: Signal Processing

We illustrate EvoX through a case study on a signal processing task (Shenoi (2005); Sharma (2025)). The goal is to build a filtering program for a noisy, changing time series. To succeed, the program must balance several competing goals: high fidelity to the signal, smoothness to reduce noise, low lag for responsiveness, and minimal false trend changes. We evaluate candidate programs using a combined score that balances these four objectives.

Figure 9 compares EvoX against a static baseline under the same 100-iteration budget. The static baseline uniformly samples the parent and inspiration set from the population and applies free-form variation to generate new candidates throughout the search. EvoX achieves a 34.1% higher final score by adaptively changing its search strategy based on observed progress, as described in Section 4.

Static Baseline. The dashed curve in Figure 9 illustrates the limitations of a fixed search strategy. This strategy achieves modest early improvement ($0.499 \rightarrow 0.530$) but then stagnates. Because the baseline chooses its parent and inspiration programs randomly and uses generic free-form variations, it mostly produces simple filters, such as basic moving averages (MA) or exponential moving averages (EMA) using NumPy. Later generations only make tiny adjustments to these simple settings, which are not enough to handle complex noise patterns. As a result, the search hits a ceiling and stays stuck.

Phase 1: Random Search and Greedy Search. EvoX begins with the same random strategy as the baseline. At iteration 20, the system detects that progress has stopped and tries a greedy strategy that focuses entirely on refining the single best program found so far. However, because the current best program still relies on simple MA/EMA structures, these tiny refinements fail to produce a breakthrough. The search remains stuck because the underlying program structure is too limited.

Phase 2: The Breakthrough (Stratified + Multi-Objective). At iteration 40, *EvoX* learns from its past failure and switches to a more advanced strategy, where instead of only selecting the best overall program, it selects parents and inspiration programs from diverse score tiers and objective-specific rankings.

For example, if a parent filter reaches a high smoothness score but has a high lag at the same time, *EvoX* selects an inspiration program that excels with lower lag. Conversely, if a parent preserves signal fidelity but leaves too much noise, it picks an inspiration program specialized in smoothing of signal. By merging these different strengths, *EvoX* discovers novel hybrid designs such as combining singular spectrum analysis (SSA) with Whittaker smoothing. This smart blending of complementary ideas creates the largest jump in performance (+0.119).

Phase 3: Structural Exploration (UCB + Structural Variation). By iteration 60, as the progress slows again, the population state descriptor shows that many recently generated candidates have similar scores. This suggests that simple refinements or combination of ideas is no longer working. To break this pattern, EvoX evolves a policy that increases the use of structural variation to attempt bolder, more complex changes.

At the same time, it uses a UCB selection rule that encourages exploration of programs that have been largely ignored (rarely selected as parents). The strategy continues to use multi-objective sampling for selecting the inspiration programs, as it proved effective in the previous search stage. It also employs the *structural variation operator* to encourage large, exploratory changes. As a result, the newly generated solutions begin to use advanced SciPy tools to construct filtering pipelines, incorporating higher-order filters, smoothing kernels, and forward-backward filtering operations (e.g., `filtfilt`). This exploration of new solution families yields a significant further improvement (+0.056).

Phase 4: Final Polishing (UCB + Local Refinement). By iteration 90, the search enters its final stage. Large structural changes now tend to destabilize performance rather than improve it. Consequently, EvoX shifts its strategy toward local refinement. This involves making small, precise adjustments to the top discovered solutions while keeping the UCB rule to prevent the search from narrowing too quickly. These fine-tuning steps provide the final gains (+0.022) and lock in the high score.

This example demonstrates the core power of *EvoX*: it changes its search strategy based on what happened in previous steps and the current variety of programs it has found. By choosing the right strategy for the right moment, EvoX avoids the plateaus that stall traditional methods and achieves substantially higher final performance.

C.2 Summary of Meta-Evolved Strategy on Other Benchmarks

Table 18: **Search strategy evolution on Signal Processing (Gemini, 100 iterations).** Initial score 0.499 $\rightarrow$ final 0.743. Δ : improvement achieved within the phase window; W : window length. Variation operator $\pi \in \{\text{local refinement}, \text{structural variation}, \text{free-form variation}\}$, where free-form variation is the default.

Strategy	Search Mechanism	Variation Operator	Δ	W
Random sampling	Parents and inspiration programs sampled uniformly from the population	Free-form variation	+0.03	11
Greedy sampling	Parents sampled from top-performing candidates	Free-form variation	+0.01	10
Stratified + multi-objective	Parents selected across objective-specific score rankings	Structural variation	+0.12	19
UCB-based selection	Usage-based parent selection with penalty for overuse	Structural variation	+0.06	16
Top-tier refinement	Parents restricted to high-score candidates	Local refinement	+0.03	14

Table 19: **Search strategy evolution on Circle Packing (Gemini, 100 iterations).** Initial score 0.364 $\rightarrow$ final 1.0004. Δ : improvement achieved within the phase window; W : window length. Variation operator $\pi \in \{\text{local refinement}, \text{structural variation}, \text{free-form variation}\}$, where free-form variation is the default.

Strategy	Search Mechanism	Operator Bias	Δ	W
Random sampling	Parents and contexts sampled uniformly from the population	Free-form variation	+0.59	21
Usage-penalized sampling	Parent selection penalizing recently sampled candidates	Structural variation	+0.04	17
Tiered sampling	Parents selected from elite, mid, and lower score bands	Mixed (refinement + structural)	+0.004	10
Stagnation-aware sampling	Parent selection conditioned on recent parent-child scores	Structural variation	+0.003	10
Elite refinement	Parents restricted to high-score candidates	Local refinement	0	20
Quantile-biased refinement	Parents sampled from upper quantiles of the score distribution	Local refinement	$\sim 2e-5$	10

Table 20: **Search strategy evolution on Heilbronn Triangle (Gemini, 100 iterations).** Initial score 0.0 $\rightarrow$ final min_area $\approx$ 0.0365 (SOTA 1.0); best at iteration 91. Δ : improvement achieved within the phase window; W : window length. Variation operator $\pi \in \{\text{local refinement}, \text{structural variation}, \text{free-form variation}\}$, where free-form variation is the default.

Strategy	Search Mechanism	Operator Bias	Δ	W
Stratified sampling	Parents selected from top and mid score bands	Free-form variation	+0.853	26
Exploration-biased sampling	Parents sampled uniformly with occasional high-score selection	Structural variation	0	10
Usage-penalized sampling	Parent selection penalizing recently sampled candidates	Mixed (refinement + structural)	+0.026	13
Tiered refinement	Parents selected from top and mid score bands	Local refinement	+0.100	16
Visit-weighted sampling	Parent selection weighted by historical sampling frequency	Free-form variation	0	10
Refinement-focused sampling	Parents sampled from high-score candidates	Local refinement	+0.021	10

D Prompt

Below is the prompt used for search strategy evolution, together with example mutation operator labels.

System Model: LLM-Driven Optimization with Evolving Search

You are an expert developer implementing a search strategy for LLM-driven program optimization. Your task is to implement EvolvedProgramDatabase, which realizes the SearchStrategy interface used by the optimizer.

The optimizer maintains a population database of evaluated candidate programs. At each step, the search strategy constructs the next-generation LLM input by choosing: (i) a parent program to modify, (ii) a variation operator that specifies how the parent

should be modified, and (iii) an optional inspiration set of past programs drawn from the population. The generated candidate is then evaluated and added back to the population.

The quality of a search strategy is determined by how effectively it improves the best solution over the course of the optimization process.

You must implement a class named `EvolvedProgramDatabase`, which defines the behavior of the search strategy through two methods:

- `add`, which controls how evaluated programs are added to the population
- `sample`, which selects the parent program, the variation operator applied to it, and an optional inspiration set of past programs.

The base class provides `self.programs: Dict[str, EvolvedProgram]` (all stored programs), `self.get(program_id)` for retrieval, and predefined label constants `self.DIVERGE_LABEL` and `self.REFINE_LABEL` (must not be modified).

Each candidate program is represented as an `EvolvedProgram`. Its fields include `id`, `code`, `metrics` (with main score `combined_score`), `iteration_found`, `timestamp`, `parent_id`, `parent_info` (a `(label, id)` tuple tracking the label applied to the parent), and `metadata`. These fields may be read but must not be modified.

The `sample` method returns a tuple of two dicts:

- `parent_dict: Dict[str, EvolvedProgram]` — maps a label to exactly one parent program;
- `inspiration_set_dict: List[EvolvedProgram]` — lists of inspirational programs providing complementary context.

Label rules. The parent label should be the empty string `""` by default. `self.REFINE_LABEL` (local refinement) or `self.DIVERGE_LABEL` (structural variation) can also be used as the parent label according to the search states.

Each generation step can only apply exactly one variation operator:

- free-form variation (the default, label `""`),
- local refinement (`REFINE_LABEL`), which requests small, structure-preserving edits,
- structural variation (`DIVERGE_LABEL`), which encourages larger, exploratory changes.

=====
CONSTRAINTS
=====

1. Program evaluation metrics are read-only and must not be modified.
2. All imports and dataclass definitions must be preserved.
3. All variables used by the strategy must be initialized with safe default values to avoid runtime errors.
4. When accessing metrics, always validate types before arithmetic (`isinstance(value, (int, float))`).

=====
CRITICAL INSTRUCTION
=====

Design a search strategy wisely and adaptively to maximize improvement.

Search evolution — end-to-end prompt (P_{search})**System message**

You are an expert software developer tasked with iteratively improving a codebase. Your objective is to maximize the **combined score** while exploring diverse solutions across feature dimensions. The system maintains a population of programs; both high performance and diversity are valuable signals.

User message

Downstream problem context. Your search algorithm evolves solutions for the following downstream problem. Use this information to inform your search strategy.

{problem_template}

Search algorithm information. **Search window.** Defines the iteration range, optimization horizon, and any adaptive replacement rules governing the lifetime of the search algorithm.

{search_window_context}

Solution population statistics. Summarizes the current state of the solution database, including population size, score distribution, stagnation signals, reuse patterns, and execution traces.

{population_state}

Current search algorithm. The program to be rewritten.

{current_program}

Relevant prior attempts. Previous search-algorithm variants that may provide useful design signals.

{relevant_programs}

Task. Rewrite the program to improve the search-algorithm score given the population state and downstream objective.

Provide the complete rewritten program and briefly explain one or two high-level principles guiding your redesign.

Keep the implementation simple, and preserve the original program's inputs and outputs while improving its internal behavior.

Your rewritten search algorithm here

E Benchmark Details

We evaluate EvoX on 196 problems: 24 optimization problems spanning three domains (mathematical optimization (8 problems), system performance (6 problems), and algorithmic challenges (10 problems)) and 172 algorithmic problems from FrontierCS.

E.1 Mathematical Optimization

Table 21: **Mathematical optimization benchmarks.** Summary of optimization tasks used to evaluate EvoX, including geometric, combinatorial, and signal processing problems.

Problem	Objective	Description
Circle Packing (Square)	$\max r$	Place n equal-radius circles inside the unit square without overlap; maximize the common radius subject to non-overlap and boundary constraints.
Circle Packing (Rectangle)	$\max r$	Pack n equal-radius circles into a rectangle with fixed aspect ratio, maximizing the radius under geometric constraints.
Heilbronn Triangle	$\max \min \text{ area}$	Place n points in $[0, 1]^2$ to maximize the area of the smallest triangle formed by any three points.
Heilbronn Convex	$\max \min \text{ area}$	Generalization of the Heilbronn triangle problem: maximize the minimum area of convex hulls formed by any subset of $k > 3$ points.
Min-Max-Min Distance (2 variants)	$\max \frac{d_{\min}}{d_{\max}}$	Place points to maximize uniformity, measured by the ratio between minimum and maximum pairwise distances.
Third Autocorrelation Inequality	$\min C_3$	Construct witness functions that minimize the third autocorrelation constant in additive combinatorics.
Signal Processing	multi-objective	Design a causal, online filtering program for noisy time series, balancing fidelity, smoothness, lag, and false trend detection.

E.2 System Performance Optimization

Table 22: **System performance optimization benchmarks.** Benchmarks drawn from ADRS-Bench (Cheng et al. (2025b)), capturing realistic optimization problems from systems.

Problem	Objective	Description
EPLB (Expert Placement Load Balancing)	$\max$ throughput ratio	Place and replicate experts in MoE models across GPUs based on activation frequency, minimizing load imbalance and computational skew.
PRISM (Global Model Scheduling)	$\max$ throughput	Schedule multiple deep learning models across distributed GPUs under memory, compute, and latency SLO constraints in a multi-tenant setting.
LLM-SQL	$\max$ accuracy	Optimize LLM-driven SQL generation via prompting or post-processing; a query is correct only if it exactly matches ground-truth execution results.
Cloudcast (Multi-Region Data Transfer)	$\min$ cost	Design cost-efficient data transfer strategies across cloud regions with heterogeneous bandwidths, latencies, and egress pricing under deadline constraints.
Transaction Scheduling	$\max$ commits/sec	Schedule database transactions under contention to maximize throughput while minimizing aborts from conflicts, deadlocks, or timeouts.
Telemetry Repair	$\max$ reconstruction accuracy	Reconstruct missing or corrupted telemetry signals using partial observations, exploiting temporal and spatial correlations.

E.3 Algorithmic and Research Problems

Table 23: **Algorithmic and research problem benchmarks.** We evaluate on 182 open-ended problems: 10 NP-hard optimization problems from ALE-Bench-Lite Imajuku et al. (2025) and 172 problems from FrontierCS Mang et al. (2025).

Problem / Category	Count	Description
<i>ALE-Bench-Lite: NP-hard optimization from AtCoder Heuristic Contests</i>		
ahc008 (Territory)	1	Control agents on a grid to place fences isolating pets; max satisfaction.
ahc011 (Sliding Puzzle)	1	Rearrange tiles so line patterns form a large connected tree; max connectivity.
ahc015 (Candy Clustering)	1	Cluster candies of the same flavor using global tilt operations; max clustering score.
ahc016 (Graph Classification)	1	Design reference graphs and classify noisy, permuted query graphs; min classification error.
ahc024 (Map Compression)	1	Compress a grid map preserving district adjacency; min map size.
ahc025 (Weight Balancing)	1	Partition items with unknown weights into balanced groups; min imbalance.
ahc026 (Box Stacking)	1	Rearrange numbered boxes across stacks to a target configuration; min moves.
ahc027 (Cleaning Route)	1	Design a cyclic robot route to minimize steady-state dirt; min average dirtiness.
ahc039 (Fishing Net)	1	Construct a rectilinear polygon enclosing targets under a perimeter budget; max net score.
ahc046 (Skating with Blocks)	1	Visit target squares in order using moves, slides, and blocks; min actions.
<i>FrontierCS: open-ended CS problems</i>		
Optimization	38	Maximize or minimize a quantitative objective over a parameterized search space under resource constraints. IDs: 1, 2, 15, 16, 22, 25, 26, 27, 28, 30, 33, 35, 36, 40, 41, 42, 43, 44, 45, 46, 47, 48, 50, 52, 53, 54, 57, 58, 59, 61, 64, 68, 69, 79, 81, 86, 93, 229.
Constructive	62	Synthesize a valid structured object (e.g., packing, graph, expression) under global constraints. IDs: 0, 3, 4, 5, 6, 7, 8, 9, 13, 17, 23, 24, 60, 62, 63, 70, 72, 73, 75, 77, 80, 82, 83, 85, 87, 89, 142, 174–193, 192, 193, 203, 205, 207, 209–214, 217, 220, 222, 225, 227, 228, 239, 241.
Interactive	72	Solve a hidden-instance task via an adaptive query-response protocol, minimizing queries or interaction steps. IDs: 10, 11, 14, 101, 104, 106–113, 117, 119–125, 127, 132–135, 137, 138, 140, 141, 143–145, 147–171, 226, 231, 233, 243, 245, 247–249, 252–258.

F Best Programs Found

We show a few representative programs found by EvoX evolutionary process.

F.1 Math

Math (Circle Packing)

```

1 # EVOLVE-BLOCK-START
2 import numpy as np
3
4 def construct_packing():
5     # Hex-core (5-4-5-4-5) + 3 top caps; multi-seed with vertical shift; short
6     ↪ SLSQP polish; uniform rescale.
7     try:
8         from scipy.optimize import minimize as mz
9     except Exception:
10        # Fallback: explicit constructor without optimization
11        R=0.1; s=np.sqrt(3)*R; y0=R
12        rows=(5,4,5,4,5)
13        core=[((R if k==5 else 2*R)+2*j*R), y0+i*s) for i,k in enumerate(rows)
14        ↪ for j in range(k)]
15        a=1-(y0+4*s); t=max(0.05,min(0.13,(a*a)/(2*(a+R)+1e-12)))
16        Y=1-t; caps=[(2*R,Y),(4*R,Y),(6*R,Y)]
17        C=np.array(core+caps,float); r=np.array([R]*23+[t]*3,float)
18        # Tighten to box
19        xm=np.min(C[:,0]-r); xM=np.max(C[:,0]+r)
20        ym=np.min(C[:,1]-r); yM=np.max(C[:,1]+r)
21        sc=min(1/((xM-xm)+1e-12),1/((yM-ym)+1e-12))
22        if sc>1.0:
23            C*=sc; r*=sc; C[:,0]-=sc*xm; C[:,1]-=sc*ym
24        return C,r,float(r.sum())
25
26 def make_seed(R,mode,shift,ks):
27     s=np.sqrt(3)*R
28     y0=max(1e-3, R-shift)
29     rows=(5,4,5,4,5)
30     core=[((R if k==5 else 2*R)+2*j*R), y0+i*s) for i,k in enumerate(rows)
31     ↪ for j in range(k)]
32     a=1-(y0+4*s)
33     t=max(0.05,min(0.13, ks*(a*a)/(2*(a+R)+1e-12))) if a>0 else 0.05
34     if mode==0: xs=(2*R,4*R,6*R)
35     elif mode==1: xs=(2*R,5*R,8*R)
36     else: xs=(1.5*R,4.5*R,8.5*R)
37     caps=[(min(max(x,t),1-t), 1-t) for x in xs]
38     C=np.array(core+caps,float); r=np.array([R]*23+[t]*3,float)
39     return C,r
40
41 def polish(C,r):
42     n=len(r)
43     v=np.empty(3*n); v[0::3]=C[:,0]; v[1::3]=C[:,1]; v[2::3]=r
44     ii,jj=np.triu_indices(n,1)
45     def obj(w): return -w[2::3].sum()
46     def c_b(w):
47         x=w[0::3]; y=w[1::3]; q=w[2::3]
48         return np.hstack((x-q,1-x-q,y-q,1-y-q))
49     def c_p(w):
50         x=w[0::3]; y=w[1::3]; q=w[2::3]
51         dx=x[ii]-x[jj]; dy=y[ii]-y[jj]; rs=q[ii]+q[jj]
52         return dx*dx+dy*dy-rs*rs
53     b=[(0,1),(0,1),(1e-6,0.5)]*n
54     res=mz(obj,v,method='SLSQP',bounds=b,
55           constraints=({'type':'ineq','fun':c_b},{'type':'ineq','fun':c_p}),
56           options={'maxiter':700,'ftol':1e-12,'disp':False})

```

```

54     w=res.x if getattr(res,'success',False) else v
55     X=np.column_stack((w[0::3],w[1::3])); q=w[2::3]
56     xm=np.min(X[:,0]-q); xM=np.max(X[:,0]+q)
57     ym=np.min(X[:,1]-q); yM=np.max(X[:,1]+q)
58     s=min(1/((xM-xm)+1e-12),1/((yM-ym)+1e-12))
59     if s>1.0:
60         X*=s; q*=s; X[:,0]-=s*xm; X[:,1]-=s*ym
61     return X,q
62
63 best_sum=-1.0; Cbest=None; rbest=None
64 for R in (0.0988,0.1000,0.1012):
65     s=np.sqrt(3)*R; a0=1-(R+4*s); sh=max(0.0, min(0.035, 0.5*a0))
66     for mode in (0,2):
67         for shift in (0.0, sh):
68             for ks in (1.0,1.06):
69                 C0,r0=make_seed(R,mode,shift,ks)
70                 C1,r1=polish(C0,r0)
71                 ssum=float(r1.sum())
72                 if ssum>best_sum:
73                     best_sum=ssum; Cbest=C1; rbest=r1
74 Cbest=np.clip(Cbest,0,1); rbest=np.maximum(rbest,0)
75 return Cbest, rbest, float(rbest.sum())
76 # EVOLVE-BLOCK-END
77
78
79 # This part remains fixed (not evolved)
80 def run_packing():
81     """Run the circle packing constructor for n=26"""
82     centers, radii, sum_radii = construct_packing()
83     return centers, radii, sum_radii
84
85
86 def visualize(centers, radii):
87     """
88     Visualize the circle packing
89
90     Args:
91     centers: np.array of shape (n, 2) with (x, y) coordinates
92     radii: np.array of shape (n) with radius of each circle
93     """
94     import matplotlib.pyplot as plt
95     from matplotlib.patches import Circle
96
97     fig, ax = plt.subplots(figsize=(8, 8))
98
99     # Draw unit square
100    ax.set_xlim(0, 1)
101    ax.set_ylim(0, 1)
102    ax.set_aspect("equal")
103    ax.grid(True)
104
105    # Draw circles
106    for i, (center, radius) in enumerate(zip(centers, radii)):
107        circle = Circle(center, radius, alpha=0.5)
108        ax.add_patch(circle)
109        ax.text(center[0], center[1], str(i), ha="center", va="center")
110
111    plt.title(f"Circle Packing (n={len(centers)}, sum={sum(radii):.6f}")
112    plt.show()
113
114
115 if __name__ == "__main__":
116     centers, radii, sum_radii = run_packing()
117     print(f"Sum of radii: {sum_radii}")
118     # AlphaEvolve improved this to 2.635

```

```

119
120     # Uncomment to visualize:
121     visualize(centers, radii)

```

Math (Min-Max-Min Distance)

```

1  # EVOLVE-BLOCK-START
2  import numpy as np
3  from scipy.optimize import minimize
4
5
6  def min_max_dist_dim3_14() -> np.ndarray:
7      """
8      Constructs 14 points maximizing min/max dist ratio using SLSQP.
9      Seeds: Rhombic Dodecahedron, Bi-capped Antiprism, Random.
10     Optimizes variable t such that  $d_{ij}^2 \geq t$  with  $d_{ij}^2 \leq 1$ .
11     """
12     n, d = 14, 3
13     iu = np.triu_indices(n, 1)
14     n_pairs = len(iu[0])
15     np.random.seed(42)
16
17     # --- Optimization Functions ---
18     def obj(x): return -x[-1]
19     def jac_obj(x):
20         g = np.zeros_like(x); g[-1] = -1; return g
21
22     def constraints(x):
23         p = x[:-1].reshape(n, d)
24         d2 = np.sum((p[iu[0]] - p[iu[1]])**2, axis=1)
25         return np.concatenate([d2 - x[-1], 1.0 - d2])
26
27     def jac_constraints(x):
28         p = x[:-1].reshape(n, d)
29         g = 2 * (p[iu[0]] - p[iu[1]])
30         J = np.zeros((2 * n_pairs, 3 * n + 1))
31         for k, (i, j) in enumerate(zip(*iu)):
32             J[k, 3*i:3*i+3] = g[k]
33             J[k, 3*j:3*j+3] = -g[k]
34             J[k, -1] = -1
35             J[n_pairs+k, 3*i:3*i+3] = -g[k]
36             J[n_pairs+k, 3*j:3*j+3] = g[k]
37         return J
38
39     # --- Seeds ---
40     seeds = []
41     # 1. Rhombic Dodecahedron (14 vertices)
42     seeds.append(np.vstack([np.eye(3), -np.eye(3),
43                             [[x,y,z] for x in (-.5,.5) for y in (-.5,.5) for z in
44                              ↪ (-.5,.5)]]))
45
46     # 2. Bi-capped Hexagonal Antiprism
47     t = np.arange(6) * np.pi / 3
48     r, h = np.sqrt(0.75), 0.5
49     seeds.append(np.vstack([
50         np.column_stack([r*np.cos(t), r*np.sin(t), np.full(6, h)]),
51         np.column_stack([r*np.cos(t+np.pi/6), r*np.sin(t+np.pi/6), np.full(6,
52                              ↪ -h)]),
53         [[0,0,1], [0,0,-1]]
54     ]))
55
56     # 3. Random
57     seeds.append(np.random.rand(n, d) - 0.5)

```

```

57 # --- Solve ---
58 best_p, best_t = None, -np.inf
59
60 for s in seeds:
61     s = s - np.mean(s, axis=0)
62     d2 = np.sum((s[iu[0]] - s[iu[1]])**2, axis=1)
63     scale = np.max(d2)
64     if scale < 1e-9: continue
65
66     x0 = np.concatenate([(s/np.sqrt(scale)).flatten(), [np.min(d2)/scale]])
67
68     try:
69         res = minimize(obj, x0, method='SLSQP', jac=jac_obj,
70                       constraints={'type': 'ineq', 'fun': constraints, 'jac':
71                                   ↪ jac_constraints},
72                       options={'maxiter': 200, 'ftol': 1e-5})
73         if -res.fun > best_t:
74             best_t = -res.fun
75             best_p = res.x
76     except: continue
77
78 if best_p is not None:
79     res = minimize(obj, best_p, method='SLSQP', jac=jac_obj,
80                   constraints={'type': 'ineq', 'fun': constraints, 'jac':
81                               ↪ jac_constraints},
82                   options={'maxiter': 1000, 'ftol': 1e-9})
83     return res.x[: -1].reshape(n, d)
84
85 return seeds[0]
86 # EVOLVE-BLOCK-END

```

F.2 System Performance Optimization

ADRS (PRISM: Global Model Scheduling)

```

1 GPU_MEM_SIZE = 80 # GB
2
3 # EVOLVE-BLOCK-START
4
5 def compute_model_placement(gpu_num, models):
6     """Minimize max KVPR via binary search on T. For target T, give each model
7     ↪ weight w=a+T*s (a=req_rate/slo, s=size) and each GPU capacity T*M; pack
8     ↪ with Best-Fit-Decreasing and return placement for minimal feasible T."""
9     M=GPU_MEM_SIZE; EPS=1e-12
10    it=[(m, m.req_rate/max(m.slo, EPS), m.model_size) for m in models]
11    if any(s>M for _,_,s in it) or sum(s for _,_,s in it)>gpu_num*M:
12        raise ValueError("Unable to place model")
13    def pack(T, ret=False):
14        cap=[T*M]*gpu_num; P=[[[] for _ in range(gpu_num)]]
15        ws=sorted([(m,a+T*s) for m,a,s in it], key=lambda x:x[1], reverse=True)
16        for m,w in ws:
17            bi=-1; br=float('inf')
18            for i in range(gpu_num):
19                r=cap[i]-w
20                if r>=-1e-12 and r<br:
21                    bi=i; br=r
22            if bi<0: return None if ret else False
23            cap[bi]-=w; P[bi].append(m)
24        return {i:P[i]} if ret else True
25    hi=1.0; tries=0
26    while not pack(hi):

```

```

25     hi*=2.0; tries+=1
26     if tries>60: raise ValueError("Unable to place model")
27     lo=0.0
28     for _ in range(35):
29         mid=(lo+hi)/2.0
30         if pack(mid): hi=mid
31         else: lo=mid
32     res=pack(hi,True)
33     if res is None: raise ValueError("Unable to place model")
34     return res
35
36 # EVOLVE-BLOCK-END
37
38
39 if __name__ == "__main__":
40     # Test the algorithm
41
42     from evaluator import generate_test_gpu_models
43     from evaluator import calculate_kvcache_pressure
44     from evaluator import safe_float
45     import numpy as np
46
47     test_cases = generate_test_gpu_models()
48     all_kvpr = []
49     for i, (gpu_num, gpu_models) in enumerate(test_cases):
50
51         results = compute_model_placement(gpu_num, gpu_models)
52         max_kvpr = calculate_kvcache_pressure(results)
53         all_kvpr.append(safe_float(max_kvpr))
54
55     avg_kvpr = np.mean(all_kvpr)
56     if avg_kvpr != 0:
57         avg_kvpr = 1.0 / avg_kvpr
58
59
60     print(f"Max KVPR: {avg_kvpr:.3f}")

```

F.3 Algorithm Engineering

ALE-Bench (AHC016: Graph Classification)

```

1 # EVOLVE-BLOCK-START
2 #include <bits/stdc++.h>
3 using namespace std;
4
5 /* Doc: Encode with deterministic block patterns. Signature = sorted expected
   ↳ noisy degrees + expected edge count.
6   Select patterns via farthest-first in this signature space; decode by nearest
   ↳ neighbor on the same signature. */
7
8 /* Squared Euclidean distance between equal-length vectors */
9 static inline double sse(const vector<double>& a, const vector<double>& b) {
10     double s = 0.0; int n = (int)a.size();
11     for (int i = 0; i < n; ++i) { double d = a[i] - b[i]; s += d * d; }
12     return s;
13 }
14
15 /* Doc: Return s positive integers that sum to N, approximately geometric with
   ↳ ratio r to separate degree levels. */
16 static inline vector<int> build_block_sizes(int N, int s, double r) {
17     vector<double> w(s); double sum = 0.0;
18     for (int i = 0; i < s; ++i) { w[i] = pow(r, i); sum += w[i]; }
19     vector<double> exact(s);

```

```

20     vector<int> base(s);
21     int total = 0;
22     for (int i = 0; i < s; ++i) { exact[i] = (double)N * w[i] / sum; base[i] =
    ↪ max(1, (int)floor(exact[i])); total += base[i]; }
23     vector<pair<double,int>> frac;
24     for (int i = 0; i < s; ++i) frac.emplace_back(exact[i] - floor(exact[i]), i);
25     sort(frac.begin(), frac.end(), greater<pair<double,int>>());
26     while (total < N) { for (int t = 0; t < s && total < N; ++t) {
    ↪ base[frac[t].second]++; total++; } }
27     while (total > N) { for (int i = s - 1; i >= 0 && total > N; --i) if (base[i] >
    ↪ 1) { base[i]--; total--; } }
28     return base;
29 }
30
31 /* Doc: Build sorted vector of expected noisy degrees under BSC noise: mu_i =
    ↪ eps*(N-1) + (1-2eps)*deg(block_i). */
32 static inline vector<double> build_mu_sorted(const vector<int>& sz, const
    ↪ vector<unsigned int>& mask, const vector<unsigned char>& diag, double eps) {
33     int s = (int)sz.size(); int N = 0; for (int v : sz) N += v;
34     vector<double> mu; mu.reserve(N);
35     const double a = eps * (N - 1), b = 1.0 - 2.0 * eps;
36     for (int bidx = 0; bidx < s; ++bidx) {
37         int d = diag[bidx] ? (sz[bidx] - 1) : 0;
38         unsigned int m = mask[bidx];
39         for (int c = 0; c < s; ++c) if (c != bidx && ((m >> c) & 1U)) d += sz[c];
40         double mv = a + b * (double)d;
41         for (int t = 0; t < sz[bidx]; ++t) mu.push_back(mv);
42     }
43     sort(mu.begin(), mu.end());
44     return mu;
45 }
46
47 /* Doc: Random symmetric block pattern: diag[b] in {0,1}, cross-block bits
    ↪ symmetric for i<j. */
48 static inline void gen_random_pattern(int s, mt19937& rng, vector<unsigned int>&
    ↪ mask, vector<unsigned char>& diag) {
49     mask.assign(s, 0U); diag.assign(s, 0);
50     uniform_int_distribution<int> bit01(0, 1);
51     for (int i = 0; i < s; ++i) diag[i] = (unsigned char)bit01(rng);
52     for (int i = 0; i < s; ++i) for (int j = i + 1; j < s; ++j) if (bit01(rng)) {
    ↪ mask[i] |= (1U << j); mask[j] |= (1U << i); }
53 }
54
55 /* Doc: Add a handful of structured patterns
    ↪ (empty/full/diag/off/banded/threshold) to diversify the pool. */
56 static inline void gen_structured_patterns(int s, vector<vector<unsigned int>>&
    ↪ masks, vector<vector<unsigned char>>& diags) {
57     auto add = [&](const vector<unsigned int>& m, const vector<unsigned char>& d){
    ↪ masks.push_back(m); diags.push_back(d); };
58     vector<unsigned int> m(s,0); vector<unsigned char> d0(s,0), d1(s,1);
59     // empty, full
60     add(m, d0);
61     vector<unsigned int> full(s,0);
62     for (int i = 0; i < s; ++i) for (int j = 0; j < s; ++j) if (i!=j) full[i] |=
    ↪ (1U<<j);
63     add(full, d1);
64     // diag-only, off-only
65     add(m, d1);
66     add(full, d0);
67     // banded |i-j|<=t
68     for (int t = 0; t < s; ++t) {
69         vector<unsigned int> mm(s,0); vector<unsigned char> dd(s, (unsigned
    ↪ char)(t&1));

```

```

70     for (int i = 0; i < s; ++i) for (int j = 0; j < s; ++j) if (i!=j &&
    ↪ abs(i-j)<=t) mm[i] |= (1U<<j);
71     add(mm, dd);
72 }
73 // threshold i+j <= T
74 for (int T = 0; T <= 2*(s-1); T += max(1,s/3)) {
75     vector<unsigned int> mm(s,0); vector<unsigned char> dd(s, (unsigned
    ↪ char)((T/2)&1));
76     for (int i = 0; i < s; ++i) for (int j = i+1; j < s; ++j) if (i + j <= T) {
    ↪ mm[i] |= (1U<<j); mm[j] |= (1U<<i); }
77     add(mm, dd);
78 }
79 }
80
81 /* Doc: Pick N as small as possible for score while ensuring enough distinct block
    ↪ patterns and noise robustness.
82 Start near a noise-dependent baseline, then increase N until capacity ~
    ↪ 2^{s(s-1)/2 + b2} >= M (or exponent >=30). */
83 static inline int chooseN(int M, double eps) {
84     auto s_of_N = [](int N){ return min(8, max(4, (int)floor(log2((double)N)) -
    ↪ 1)); };
85     // noise-aware baseline with small-N bias at low eps
86     int N = (int)llround(10.0 + 100.0 * eps);
87     int Nmin = (eps < 0.15 ? 10 : (eps < 0.28 ? 14 : 18));
88     N = max(N, Nmin);
89     N = min(N, 100);
90     double r = max(1.35, min(2.35, 1.5 + 1.5 * eps));
91     while (true) {
92         int s = s_of_N(N);
93         auto sz = build_block_sizes(N, s, r);
94         int b2 = 0; for (int v : sz) if (v >= 2) ++b2;
95         int expo = s*(s-1)/2 + b2;
96         if (expo >= 30 || (1 << expo) >= M) break;
97         if (N >= 100) break;
98         ++N;
99     }
100     if (M > 80 && eps > 0.25) N = min(100, N + 2); // stabilize in high-noise/high-M
101     return max(8, N);
102 }
103
104 /* Build adjacency bitstring from block pattern and vertex->block mapping */
105 static inline string build_graph_bits(int N, const vector<pair<unsigned
    ↪ char,unsigned char>>& pairs,
106                                     const vector<int>& belong, const
    ↪ vector<unsigned int>& mask, const
    ↪ vector<unsigned char>& diag) {
107     int L = N * (N - 1) / 2; string s(L, '0');
108     for (int p = 0; p < L; ++p) {
109         auto ab = pairs[p];
110         int bu = belong[ab.first], bv = belong[ab.second];
111         bool conn = (bu == bv) ? (diag[bu] != 0) : (((mask[bu] >> bv) & 1U) != 0);
112         if (conn) s[p] = '1';
113     }
114     return s;
115 }
116
117 int main() {
118     ios::sync_with_stdio(false);
119     cin.tie(nullptr);
120
121     int M; double eps;
122     if (!(cin >> M >> eps)) return 0;
123
124     // Tune block-size ratio by noise level to widen degree gaps as noise increases
125     double BLOCK_R = max(1.35, min(2.35, 1.5 + 1.5 * eps));

```

```

126
127 // Choose N and s (number of blocks)
128 int N = chooseN(M, eps);
129 int s = min(8, max(4, (int)floor(log2((double)N)) - 1 + (eps < 0.08 ? 1 : 0) -
    ↪ (eps > 0.28 ? 1 : 0)));
130 vector<int> sz = build_block_sizes(N, s, BLOCK_R);
131
132 // Vertex -> block mapping
133 vector<int> belong(N, 0);
134 for (int i = 0, cur = 0, b = 0; b < s; ++b) for (int t = 0; t < sz[b]; ++t)
    ↪ belong[cur++] = b;
135
136 // Precompute (i,j) pairs and constants
137 int L = N * (N - 1) / 2;
138 vector<pair<unsigned char, unsigned char>> pairs;
139 pairs.reserve(L);
140 for (int i = 0; i < N; ++i) for (int j = i + 1; j < N; ++j)
141     pairs.emplace_back((unsigned char)i, (unsigned char)j);
142
143 // Variance-normalized weights for fused scoring (degrees + edge count)
144 double invVarDeg = 1.0 / ((double)(N - 1) * eps * (1.0 - eps) + 1e-12);
145 double invVarM = 1.0 / ((double)L * eps * (1.0 - eps) + 1e-12);
146 double degW = 1.0 + 0.8 * eps;
147
148 // Build candidate pool of block patterns (structured + random + complements)
149 int seed = 146527 + M * 1000 + (int)llround(eps * 100.0) * 7919;
150 mt19937 rng(seed);
151 vector<vector<unsigned int>> cand_masks;
152 vector<vector<unsigned char>> cand_diags;
153
154 gen_structured_patterns(s, cand_masks, cand_diags);
155
156 auto add_comp = [&](const vector<unsigned int>& m, const vector<unsigned
    ↪ char>& d){
157     vector<unsigned int> mc(s, 0u); vector<unsigned char> dc(s, 0);
158     unsigned int all = (s >= 31 ? 0x7FFFFFFFu : ((1u << s) - 1u));
159     for (int i = 0; i < s; ++i) { unsigned int fulli = all & ~(1u << i); mc[i]
    ↪ = fulli ^ m[i]; dc[i] = (unsigned char)(1 - d[i]); }
160     cand_masks.push_back(mc); cand_diags.push_back(dc);
161 };
162
163 int RAND_CANDS = (eps <= 0.12 ? max(6 * M, 384) : min(3072, max(10 * M, 512)));
164 for (int t = 0; t < RAND_CANDS; ++t) {
165     vector<unsigned int> m; vector<unsigned char> d;
166     gen_random_pattern(s, rng, m, d);
167     cand_masks.push_back(m); cand_diags.push_back(d);
168     add_comp(m, d);
169 }
170
171 struct Cand { vector<unsigned int> mask; vector<unsigned char> diag;
    ↪ vector<double> mu; double mu_m; };
172 vector<Cand> pool; pool.reserve(cand_masks.size());
173 for (size_t i = 0; i < cand_masks.size(); ++i) {
174     vector<double> mu = build_mu_sorted(sz, cand_masks[i], cand_diags[i], eps);
175     long long edges = 0;
176     for (int b = 0; b < s; ++b) if (cand_diags[i][b]) edges += 1LL * sz[b] *
    ↪ (sz[b] - 1) / 2;
177     for (int b = 0; b < s; ++b) for (int c = b + 1; c < s; ++c)
178         if ((cand_masks[i][b] >> c) & 1U) edges += 1LL * sz[b] * sz[c];
179     double mu_m = eps * (double)L + (1.0 - 2.0 * eps) * (double)edges;
180     pool.push_back({cand_masks[i], cand_diags[i], move(mu), mu_m});
181 }
182
183 // Greedy farthest-point sampling using fused metric on (mu_sorted, mu_m)
184 auto fused_dist = [&](int a, int b)->double{

```

```

185     double d = sse(pool[a].mu, pool[b].mu) * invVarDeg * degW;
186     double dm = pool[a].mu_m - pool[b].mu_m;
187     return d + dm * dm * invVarM;
188 };
189 vector<int> chosen;
190 int C = (int)pool.size();
191 int seedIdx = 0; double bestVar = -1.0;
192 for (int i = 0; i < C; ++i) {
193     const auto& mu = pool[i].mu;
194     double mean = 0.0; for (double v : mu) mean += v; mean /= mu.size();
195     double var = 0.0; for (double v : mu) { double d = v - mean; var += d * d; }
196     if (var > bestVar) { bestVar = var; seedIdx = i; }
197 }
198 chosen.push_back(seedIdx);
199 for (int k = 1; k < M; ++k) {
200     int nxt = -1; double far = -1.0;
201     for (int i = 0; i < C; ++i) {
202         bool used = false; for (int id : chosen) if (id == i) { used = true;
203             ↪ break; }
204         if (used) continue;
205         double md = 1e300;
206         for (int id : chosen) md = min(md, fused_dist(i, id));
207         if (md > far) { far = md; nxt = i; }
208     }
209     if (nxt < 0) nxt = chosen.back();
210     chosen.push_back(nxt);
211 }
212 // Output graphs
213 cout << N << '\n';
214 vector<vector<double>> mu_sorted_vec(M, vector<double>(N, 0.0));
215 vector<double> mu_m_vec(M, 0.0);
216 for (int k = 0; k < M; ++k) {
217     const auto& pat = pool[chosen[k]];
218     mu_sorted_vec[k] = pat.mu; mu_m_vec[k] = pat.mu_m;
219     string sbits = build_graph_bits(N, pairs, belong, pat.mask, pat.diag);
220     cout << sbits << '\n';
221 }
222 cout.flush();
223
224 // Online decoding: fused nearest neighbor using degree multiset + edge count
225 for (int q = 0; q < 100; ++q) {
226     string H; if (!(cin >> H)) return 0;
227     vector<int> deg(N, 0); int m = 0;
228     for (int p = 0; p < L; ++p) if (H[p] == '1') { auto ab = pairs[p];
229         ↪ deg[(int)ab.first]++; deg[(int)ab.second]++; ++m; }
230     vector<double> dh(N); for (int i = 0; i < N; ++i) dh[i] = (double)deg[i];
231     sort(dh.begin(), dh.end());
232     int best = 0; double bestS = 1e300;
233     for (int k = 0; k < M; ++k) {
234         double sdeg = sse(dh, mu_sorted_vec[k]) * invVarDeg * degW;
235         double dm = (double)m - mu_m_vec[k];
236         double score = sdeg + dm * dm * invVarM;
237         if (score < bestS) { bestS = score; best = k; }
238     }
239     cout << best << '\n';
240     cout.flush();
241 }
242 return 0;
243 }
244 # EVOLVE-BLOCK-END

```

G Configurations for Different Frameworks

We present the configurations we use for evaluating open-source evolutionary frameworks including OpenEvolve, GEPA, and ShinkaEvolve.

GEPA Configuration

```

1 # Optimization budget
2 max_metric_calls: 50
3 reflection_minibatch_size: 1
4
5 # Model
6 reflection_lm: "gpt-5"
7
8 # Adapter
9 # One of: circle_packing, signal_processing, heilbronn_convex, etc.
10 adapter: "circle_packing"
11 timeout: 360

```

GEPA Reflection Prompt

Problem context

{problem_context}

Current code

Here is the current Python code for this problem:

<curr_instructions>

Evaluation results

Below are the results from running this code, including scores and feedback on how to improve:

<inputs_outputs_feedback>

Your task is to write an IMPROVED VERSION of the Python code above.

Critical requirements.

1. You **must** output valid, executable Python code, not natural language descriptions or instructions.
2. The code must be self-contained with all necessary imports (numpy, jax, optax, scipy, etc.).
3. The code must define a run() function as the entry point that returns the expected output format.
4. Keep the same return signature as the original code.
5. Include # EVOLVE-BLOCK-START at the beginning and # EVOLVE-BLOCK-END at the end.
6. Focus on algorithmic improvements: better initialization, optimization strategies, hyperparameters, or mathematical approaches.
7. Do **not** output explanations outside the code block. Only output the code.

Analyze the feedback carefully. The score indicates how close the current solution is to the benchmark (1.0 = matches benchmark). Identify what can be improved and implement those changes in the code.

Your improved Python code here

OpenEvolve Configuration

```

1 # Evolution settings
2 max_iterations: 100
3 checkpoint_interval: 10
4 parallel_evaluations: 1
5
6 # LLM configuration
7 llm:
8   primary_model: "gpt-5"
9   primary_model_weight: 1.0
10  temperature: 0.7
11  top_p: 0.95
12  max_tokens: 32000
13  timeout: 600
14
15 # Database configuration (MAP-Elites)
16 database:
17   population_size: 40
18   num_islands: 5
19   migration_interval: 40
20   feature_dimensions: # MUST be a list
21     - "score"
22     - "complexity"
23
24 # Evaluation settings
25 evaluator:
26   timeout: 360
27   max_retries: 3
28
29 # Prompt configuration
30 prompt:
31   num_top_programs: 3
32   num_diverse_programs: 2

```

ShinkaEvolve Configuration

```

1 # Shinka evolution configuration (fixed parent-selection strategy)
2
3 # Evolution budget
4 max_iterations: 100
5 max_parallel_jobs: 4
6
7 # LLM configuration
8 llm:
9   primary_model: "gpt-5"
10  temperatures: [0.0, 0.7, 1.0]
11  max_tokens: 32768
12  reasoning_efforts: ["auto"]
13
14 # Patch generation
15 patch_types: ["diff", "full", "cross"]
16 patch_type_probs: [0.6, 0.3, 0.1]
17 max_patch_resamples: 3
18 max_patch_attempts: 3
19
20 # Database / archive configuration
21 database:
22   num_islands: 5
23   archive_size: 20
24
25 # Parent selection (fixed strategy)
26 parent_selection_strategy: "weighted"
27 parent_selection_lambda: 10.0
28

```

```
29 # Archive inspiration
30 elite_selection_ratio: 0.3
31 num_archive_inspirations: 4
32 num_top_k_inspirations: 2
33
34 # Island migration
35 migration_interval: 5
36 migration_rate: 0.1
37 island_elitism: true
38
39 # Meta-reasoning (non-adaptive)
40 meta:
41   meta_rec_interval: 10
42   meta_model: "gpt-5"
43   meta_temperature: 0.7
44   meta_max_tokens: 16384
45
46 # Novelty estimation
47 novelty:
48   embedding_model: "text-embedding-3-small"
49   code_embed_sim_threshold: 0.995
50
51 # LLM routing
52 llm_dynamic_selection: "ucb1"
53 llm_dynamic_selection_kwargs:
54   exploration_coef: 1.0
```