EVADE: EVOLVABILITY-GUIDED ALGORITHMIC DISCOVERY

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) are increasingly used as mutation operators in evolutionary search, enabling the discovery of novel algorithms for problems in mathematics, computer systems, and other domains. Yet existing LLM-based frameworks largely inherit the classical evolutionary paradigm of selecting candidates for mutation based on quality and diversity. These criteria capture the current state of individual candidates—such as their fitness (quality) or how they differ from other selected candidates (diversity)—but do not capture their potential for further evolution. We introduce *evolvability* as a search signal: a candidate's capacity to generate offspring that improve fitness or exhibit behaviors that differ from those of its ancestors. We then propose EVADE, which uses the estimated evolvability score to guide candidate selection, exploration–exploitation, and mutation context. Across mathematical and systems optimization problems, EVADE reduces redundant search and achieves the best mean and best solution on most tasks. Case studies demonstrate EVADE's ability to escape from poor solutions and discover an algorithm that surpasses the human state of the art on a real-world cloud data transfer task. Our code is available for reproducibility.[1]

1 INTRODUCTION

Richard Sutton articulated the *Bitter Lesson (2019): "AI progress has largely come from methods that leverage computation rather than explicitly encoding human knowledge."* Sutton identified learning and search as two fundamental ways to effectively leverage computation, allowing systems to improve from experience and explore a large space of possible solutions. He illustrated this with chess and Go examples, where large-scale search ultimately outperformed approaches based on extensive efforts to structure and encode human knowledge. While the two approaches need not conflict, he noted that "the time that is spent on one is time not spent on the other."

But what if encoding and leveraging human knowledge can themselves be viewed as learning and search problems, making them scalable with computation? LLM-based evolutionary computation (LLM-based EC) offers a concrete example of this possibility. Through large-scale training, LLMs encode vast amounts of human knowledge, with their capabilities scaling with data and compute. We can then leverage this knowledge at test time to generate ideas, translate them into executable programs, and iteratively search for improvements. By scaling test-time computation, LLM-based EC can explore increasingly large solution spaces and discover new programs for solving difficult problems (Snell et al., 2025), opening a new mechanism for accelerating AI progress. This paradigm has shown promising results, including mathematical problem solving (Romera-Paredes et al., 2023), systems performance optimization (Novikov et al., 2025; Cheng et al., 2025a;b), and heuristic algorithm design (Liu et al., 2024; Ye et al., 2024; Hu & Zhang, 2025).

LLM-based EC systems typically maintain an archive of executable programs. At each step, a candidate, called the *parent*, is selected and passed to an LLM, which generates a mutated *child* program conditioned on a specific guideline prompt. The resulting *child* program is then evaluated to obtain its fitness. The system goal is to search the solution space for programs that maximize the target fitness under a fixed budget, as each step incurs both an LLM call and an evaluation cost. The key decisions in such systems are therefore *when to explore or exploit, which parent to mutate,*

[1] `https://anonymous.4open.science/r/evade-A2E0/README.md`

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

and how to guide the mutation. Existing systems primarily make these decisions based on program quality and diversity (Novikov et al., 2025; Sharma, 2025; Lange et al., 2026; Cemri et al., 2026).

These systems, however, do not explicitly account for *evolvability* (Altenberg et al., 1994; Wagner & Altenberg, 1996), which measures whether a candidate can still evolve into solutions that are novel or better than those already found. Quality and diversity characterize a candidate's current state, but neither captures its remaining evolutionary potential. Consequently, search guided solely by these criteria may repeatedly select candidates whose useful improvements have already been exhausted or generate programs that differ syntactically but exhibit similar behaviors, wasting the search budget in regions that have already been explored.

We propose EVADE, an LLM-guided evolutionary search framework that derives an empirical evolvability signal from the outcomes of the mutations generated from each program. The signal captures whether these mutations surpassed the best fitness found so far or reached behaviors outside the parent's ancestral lineage, and updates the evolvability score of the parent's behavioral cell. EVADE uses this signal to determine when to explore or exploit, which parent to mutate, and which past experience to provide to the LLM. Because it is derived from mutations already performed, the signal requires no additional evaluations. Our contributions are as follows:

- We introduce an evolvability signal for LLM-guided evolutionary search, computed online from the mutations the search already performs, without additional evaluations.
- We propose EVADE, which uses this signal to guide the balance between exploration and exploitation, parent selection, and the experience supplied to each mutation.
- We evaluate EVADE on 11 mathematical and systems optimization tasks with two backbone LLMs, where it achieves the best mean and best solution on most tasks. Ablations show that these gains come from allocating the budget toward parents that remain evolvable and from guiding the LLM with suitable experience. Case studies further show that EVADE reduces redundant search, escapes deep local optima, and surpasses the human state of the art on a systems optimization task.

2 RELATED WORK

LLM-based Evolutionary Computation LLM-based EC inherits the classical evolutionary loop of maintaining solutions, evaluating fitness, selecting candidates, and generating new solutions through variation. Its key differences lie in **candidate representation** and **variation operators** (Lehman et al., 2023; Romera-Paredes et al., 2023; Liu et al., 2024). Candidates are represented as natural-language ideas paired with executable programs, while hand-crafted mutation and crossover are replaced by LLM-driven generation guided by prompts (Cemri et al., 2026; Assumpção et al., 2025; Lange et al., 2026; Ye et al., 2024; Jiang et al., 2026). Recent work further uses evolutionary experience to fine-tune LLM for solution generation (Wang et al., 2026; Yuksekgonul et al., 2026).

Existing frameworks also differ in *population maintenance* and *candidate selection*. For population maintenance, MAP-Elites archives (Mouret & Clune, 2015) preserve elites across behavioral cells, while island-based methods (Cantu-Paz, 2000) evolve semi-independent populations with periodic migration. ELM (Lehman et al., 2023) uses MAP-Elites; FunSearch (Romera-Paredes et al., 2023), ShinkaEvolve (Lange et al., 2026), CodeEvolve (Assumpção et al., 2025), and AdaEvolve (Cemri et al., 2026) use islands; and AlphaEvolve (Novikov et al., 2025) and OpenEvolve (Sharma, 2025) combine both. PartEvo (Hu & Zhang, 2025) instead groups programs by code and thought similarity. For candidate selection, early methods favor high fitness (Romera-Paredes et al., 2023; Liu et al., 2024), while later approaches balance fitness and diversity (Novikov et al., 2025; Sharma, 2025; Agrawal et al., 2026; Lange et al., 2026; Cemri et al., 2026; Liu et al., 2026a).

Evolvability has long been recognized as an important ingredient in EC, yet still an underexplored concept, particularly in combination with LLMs. Early studies Altenberg et al. (1994); Wagner & Altenberg (1996) identified evolvability as a key factor underlying evolutionary success or failure. Unlike quality and diversity, which describe a candidate itself (how well it performs and how much it differs from others), evolvability describes what its offspring achieve. Subsequent work showed that evolvability can emerge as a side effect of representation design, developmental processes, drift, and novelty pressures Doncieux et al. (2020); Wagner & Altenberg (1996); Lehman & Stanley (2013). Later approaches began to model evolvability directly, typically by measuring the

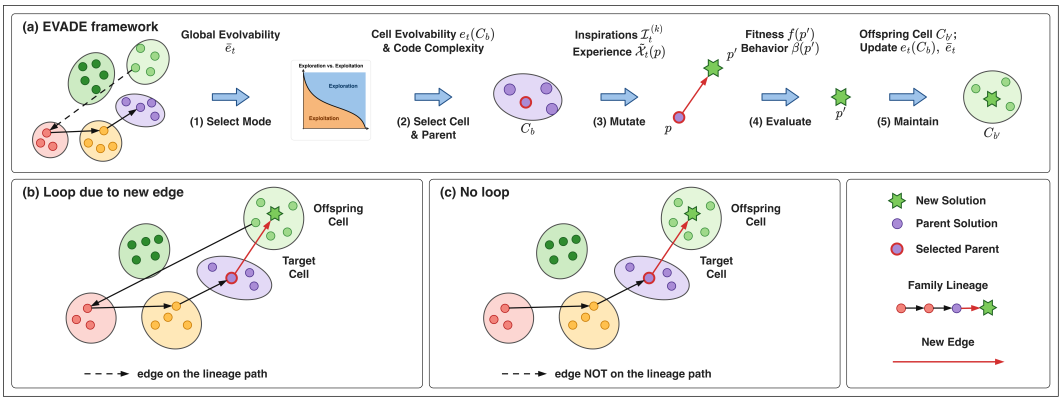


Figure 1: **Overview of EVADE. (a)** The five steps of one iteration. **(b, c)** The offspring cell determines the evolvability signal of the selected parent: **(b)** returning to a cell on the parent's ancestral lineage closes a loop and yields a signal of 0, unless the new solution surpasses the best fitness found so far; **(c)** reaching a cell outside the lineage yields a positive signal.

diversity or novelty of offspring behaviors (Mengistu et al., 2016; Gajewski et al., 2019), survival rate (Cully & Demiris, 2017), gradient estimation, or Markov models Katona et al. (2021); Ferigo et al. (2022); Gasperov & Durasevic (2023).

Existing methods, however, measure a candidate's evolvability without considering its ancestral lineage. We instead define evolvability as a candidate's capacity to generate offspring that either improve fitness or exhibit behaviors different from its ancestors, capturing phenotypic change across its family's generations. We further aggregate individual evolvability to characterize the evolvability of populations and behavioral cells, and use it as a control signal to guide the exploration–exploitation trade-off, as well as mutation guidelines for LLMs. Unlike previous works in EC Mengistu et al. (2016); Gajewski et al. (2019); Cully & Demiris (2017); Katona et al. (2021); Ferigo et al. (2022); Gasperov & Durasevic (2023), where evolvability is a selection criterion, an objective to optimize or an outcome of exploration strategies, we make evolvability a direct driver of evolution.

3 THE EVADE FRAMEWORK

3.1 PROBLEM FORMULATION

We formalize LLM-guided program search as maximizing a fitness function $f : \mathcal{P} \to \mathbb{R}$ over a discrete space of executable programs $\mathcal{P}$, subject to a computational budget of T iterations. The search maintains an *archive* $A_t \subset \mathcal{P}$ of all programs evaluated up to iteration t. Each program $p \in A_t$ is stored with its fitness $f(p)$ and a behavior descriptor $\beta(p)$ capturing its execution profile under the evaluator. Following Agrawal et al. (2026), $\beta(p)$ can simply be the scalar fitness itself, a vector of trade-off metrics in multi-objective tasks (e.g., latency, cost, and accuracy), sub-scores across test cases, or any domain-specific measurements tailored to the problem.

The archive is partitioned into *cells* $\mathcal{C}_t = \{C_b\}_{b=1}^{K_t}$ by behavioral similarity, grouping programs with similar descriptors. The number of cells K_t is dynamic: new cells emerge as novel behaviors appear, and the entire partition is recomputed periodically as the archive expands. Each program also stores a pointer to its parent, which links cells into separate ancestral lineages: the ancestral lineage $\mathcal{B}(p)$ of a program p is the set of cells occupied by p and its ancestors. When a child p' derived from $p \in C_b$ is assigned to a cell $C_{b'}$, C_b becomes an ancestor of $C_{b'}$, and $\mathcal{B}(p') = \mathcal{B}(p) \cup \{C_{b'}\}$. Each cell C_b tracks an *evolvability score* $e_t(C_b) \in [0, 1]$, which estimates the likelihood that mutating programs in C_b will discover unexplored behavioral regions or yield fitness improvements. Aggregating these scores across cells yields a *global evolvability score* $\bar{e}_t$.

The evolutionary cycle of EVADE is illustrated in Figure 1. Each iteration consists of five steps: (1) **Mode selection**: use global evolvability to select among *exploit*, *explore*, *balance*, and *breakthrough* modes; (2) **Selection**: sample a cell $C_b \in \mathcal{C}_t$ based on its evolvability and fitness, then select a

parent $p \in C_b$ based on code complexity; (3) **Mutation**: prompt an LLM to generate a child p' from p using *inspiration programs* from A_t as in-context exemplars and *experience* of successful and unsuccessful modifications. Local experience records mutation strategies tried in each cell and their score changes, with p receiving experience from all cells in its ancestral lineage $\mathcal{B}(p)$; global experience records recent fitness-improving and degrading edits across the run; (4) **Evaluation**: evaluate p' to obtain its fitness $f(p')$ and behavior descriptor $\beta(p')$; and (5) **Maintenance**: update the parent's cell evolvability, global evolvability, and periodically recluster the behavioral cells.

3.2 Evolvability Estimation

Evolvability Signal. *Each mutation provides evidence about how evolvable its parent is.* At iteration t, a parent $p \in C_b$ produces a child p', which lands in the cell $C_{b'}$. We record the *evolvability signal* $r_t(p) \in [0, 1]$ of p at iteration t:

$$r_t(p) = \begin{cases} 1 & \text{if } f(p') > f_t^\star, \\ e_{t-1}(C_{b'}) \left[\alpha + (1-\alpha)\,\tilde{\delta}_t\right] & \text{if } f(p') \le f_t^\star \text{ and } C_{b'} \notin \mathcal{B}(p), \\ 0 & \text{if } f(p') \le f_t^\star \text{ and } C_{b'} \in \mathcal{B}(p), \end{cases} \tag{1}$$

where $f_t^\star$ is the global best fitness before iteration t. The signal operationalizes our desiderata for productive evolution: i) a child that surpasses the global best receives the maximum signal wherever it lands; ii) a child that returns to a cell its own lineage has already occupied receives none, since it reaches no new behavior compared to its ancestors.

Given the hyperparameter α ($\alpha = 0.5$ by default; sensitivity in Table 5), the value of $r_t(p)$ in the remaining case depends on two terms: (1) $e_{t-1}(C_{b'})$ is the evolvability of the child cell $C_{b'}$; (2) $\tilde{\delta}_t$ is the relative improvement, which is measured as follows:

$$\tilde{\delta}_t = \min\left(1,\ \frac{\delta_t}{\bar{\delta}_t}\right), \qquad \delta_t = \max\big(0,\ f(p') - f(p)\big), \tag{2}$$

where δ_t indicates the gain of p' over its parent p, and $\bar{\delta}_t$ estimates the improvement the search achieves in its recent steps,

$$\bar{\delta}_t = \begin{cases} \rho\,\bar{\delta}_{t-1} + (1-\rho)\,\delta_t & \text{if } \delta_t > 0, \\ \bar{\delta}_{t-1} & \text{otherwise,} \end{cases} \tag{3}$$

with decay factor $\rho \in [0, 1)$, set to 0.9 (sensitivity in Table 5). Depending on whether $\tilde{\delta}_t < 1$ or $\tilde{\delta}_t = 1$, the evolvability signal of the parent p is either smaller than or equal to the evolvability score of the child (p') cell. Note that, $\tilde{\delta}_t$ is 1 when the child gain is at least the historical base value $\bar{\delta}_t$.

Cell Evolvability. Cell evolvability $e_t(C_b)$ is computed from the programs assigned to it. Whenever $p \in C_b$ is selected as a parent and receives an evolvability signal $r_t(p) \in [0, 1]$, we update the discounted signal sum and the visit count associated with p as follows:

$$S_p \leftarrow \rho\, S_p + r_t(p), \qquad N_p \leftarrow \rho\, N_p + 1, \tag{4}$$

with ρ the decay factor of Eq. equation 3. The decay gives recent outcomes more weight, and decaying N_p alongside S_p keeps their ratio a properly normalized weighted average of signals.

The cell signal sum S_b and visit count N_b are obtained by summing S_p and N_p of all programs in C_b, $S_b = \sum_{p \in C_b} S_p$ and $N_b = \sum_{p \in C_b} N_p$, and its evolvability score is their smoothed ratio,

$$e_t(C_b) = \frac{S_b + \kappa\, \bar{e}_{t-1}}{N_b + \kappa}, \tag{5}$$

where $\bar{e}_{t-1}$ is the global evolvability score from the preceding iteration (Eq. equation 6) and $\kappa = 4$. This form has three properties. It is bounded in $[0, 1]$, since $S_b \le N_b$. It is defined for a cell that has never been selected ($S_b = N_b = 0$), which takes the value $\bar{e}_{t-1}$. And κ controls a cell's score from moving sharply over its first few attempts. The sensitivity to κ is in Table 5.

Global Evolvability. The global evolvability score is the visit-weighted mean over cells,

$$\bar{e}_t = \frac{\sum_b N_b\, e_t(C_b)}{\sum_b N_b}, \tag{6}$$

taking the initial value of 0.5. It is computed once per iteration after updating all cell scores and exploited as the prior in Eq. 5 for iteration $t+1$. Weighting by N_b prevents rarely attempted cells from dominating, $\bar{e}_t$ tracks recent mutation productivity across cells and decides the selection mode.

3.3 EVOLVABILITY-GUIDED SELECTION

We first sample a cell balancing quality and evolvability, then a parent based on complexity.

Mode Selection. Inspired by AdaEvolve, we adapt the exploration–exploitation trade-off through four modes: *exploit*, which favors high-fitness cells and nearby parent variants; *balance*, which weighs fitness and evolvability equally; *explore*, which favors highly evolvable cells; and *breakthrough*, which favors fitness while drawing inspiration from top candidates across cells. Unlike AdaEvolve, EVADE samples a mode k from a distribution $\Pr(k)$ driven by the global evolvability (Eq. 6). Details are provided in Appendix A.2.

Cell Selection. EVADE samples cells using a mode-specific fitness–evolvability trade-off. Specifically, given the mode k, each cell receives a utility combining fitness and evolvability:

$$u(C_b) = w_f^{(k)}\,\hat{f}(C_b) + w_e^{(k)}\,e_t(C_b), \tag{7}$$

where $\hat{f}(C_b) \in [0,1]$ is the mean fitness of the cell's members, min–max normalized across all occupied cells to match the scale of $e_t(C_b)$, and $w_f^{(k)}$ and $w_e^{(k)}$ are the weights of mode k, fixed across all tasks (Table 4, Appendix A.2). Inspired by *score-proportionate selection* (Cully & Demiris, 2017), we prioritize cells with high utility $u(C_b)$, sampling each cell with probability $\Pr(C_b) = u(C_b)/\sum_{C\in\mathcal{C}_t} u(C)$.

Parent & Inspiration Program Selection. Parent programs are sampled from the selected cell based on their complexity (McCabe, 1976). This is motivated by the observation that more complex candidates are harder to mutate productively, a failure mode also reported in automated discovery practice (Cheng et al., 2025a). Details on how program complexity is computed are provided in Appendix A.2. Following (Cemri et al., 2026; Lange et al., 2026), we supply $|\mathcal{I}_t^{(k)}| = 4$ inspiration programs selected according to the mode: exploit and balance draw them from the parent's own cell, while explore and breakthrough draw the best program of each other cell (Appendix A.2).

3.4 EVOLVABILITY-GUIDED MUTATION

Experience Control via Evolvability. For a parent $p \in C_b$, let $\mathcal{X}_t(p)$ denote the experience available at iteration t: the global experience and the local experience of the cells in $\mathcal{B}(p)$ (§3.5). Experiences are updated after each evolutionary cycle based on the evaluation outcomes.

To avoid over-saturating the mutation prompt with redundant or stale guidance, older experience is progressively filtered out rather than kept in full, following the adaptive memory principle of tabu search (Battiti & Tecchiolli, 1994). Specifically, entries are ranked by recency, and the i-th rank entry at time t in cell C_b is retained with probability $\lambda_t(C_b)^i$. Here, $\lambda_t(C_b)$ is the survival factor for cell C_b, computed based on the cell evolvability as follows:

$$\lambda_t(C_b) = \lambda_{\min} + \big(\lambda_{\max} - \lambda_{\min}\big)\,e_t(C_b), \tag{8}$$

where $\lambda_{\min} = 0.5$ and $\lambda_{\max} = 0.85$ bound the survival factor, yielding about one to five retained programs per cell in expectation. Exponential decay favors recent entries while keeping the expected number bounded, preventing LLM context growth.

The entries that survive this filtering form the retained experience $\tilde{\mathcal{X}}_t(p) \subseteq \mathcal{X}_t(p)$, which is supplied to the mutation operator. As $e_t(C_b)$ drops, $\lambda_t(C_b)$ contracts and aggressively prunes older feedback. Intuitively, we restrict old guidance so that the LLM is not distracted by exhausted local modifications from cells with decreasing evolvability.

Mutation Operator. Given a parent p drawn from cell C_b under mode k, together with the inspiration programs $\mathcal{I}_t^{(k)}$ (§3.3) and the retained experience $\tilde{\mathcal{X}}_t(p)$ (Eq. equation 8), the LLM first generates a mutation strategy g and then produces a child solution p':

$$\Pr\big(p' \mid p, \tilde{\mathcal{X}}_t(p), \mathcal{I}_t^{(k)}\big) = \sum_g \pi_k^g\big(g \mid p,\, \tilde{\mathcal{X}}_t(p),\, \mathcal{I}_t^{(k)}\big)\;\cdot\;\pi_k^c\big(p' \mid p,\, g,\, \tilde{\mathcal{X}}_t(p),\, \mathcal{I}_t^{(k)}\big), \tag{9}$$

where π_k^g, π_k^c are the strategy and program policies under mode k. We first generate g then p' in one LLM call. After evaluation, g and its score change are saved as the local experience of p's cell.

3.5 MAINTENANCE

EVADE maintains two components: the behavioral cell partition and the experience bank.

Behavioral Cell Maintenance. We organize the behavioral space into discrete cells $\mathcal{C}_t$, grouping programs with similar behavior while separating distinct behaviors. When a new program p' is evaluated, ASSIGNCELL (Alg. 5) assigns it to the nearest cell if its normalized distance is within the merge distance threshold ϵ; otherwise, a new cell is created. To reduce dependence on evaluation order, EVADE periodically rebuilds the partition using RECLUSTER (Alg. 6) every τ_{cell} iterations. Unlike code-partitioning strategies (Hu & Zhang, 2025), we cluster programs by runtime behavior rather than static code semantics. RECLUSTER normalizes behavioral descriptors, adaptively sets ϵ to the 25th percentile of pairwise distances, performs agglomerative clustering, reconstructs ancestral lineages from parent pointers, and updates cell evolvability scores $e_t(C_b)$ via Eq. equation 5.

Experience Maintenance. Accumulating qualitative feedback is common in evolutionary LLM search (Ye et al., 2024; Lange et al., 2026; Cemri et al., 2026; Jiang et al., 2026). EVADE maintains it at two levels (Alg. 7). Local experience records, for each cell, the strategies already tried from it and their score changes, so that the LLM avoids repeating them. Global experience is updated every τ_{exp} iterations: an LLM describes what each of the recent edits did, and these descriptions are stored separately for edits that improved and edits that degraded fitness.

4 EXPERIMENTS

Tasks and Baselines. We evaluate EVADE on 11 optimization problems: 6 mathematical optimization problems (§4.1) and 5 ADRS systems optimization problems (§4.1), all drawn from the SkyDiscover suite (Liu et al., 2026b). We compare EVADE against evolutionary algorithms with open-source codebases: GEPA, ShinkaEvolve, AdaEvolve, and EvoX (§2).

Setup. All methods are evaluated with the same backbone LLMs, GLM-5.2 (GLM-5-Team et al., 2026) and DeepSeek-V4-Flash (DeepSeek-AI, 2026), the same predefined evaluator for fitness scoring in each task, and a budget of $T = 100$ iterations. We report the mean $\pm$ standard deviation over three independent runs, along with the best of these three runs, following prior work (Cemri et al., 2026; Liu et al., 2026a). While domain-specific behavior descriptors may benefit individual tasks, we use a unified combined score—defined as a single objective or aggregate of multiple objectives—to enable broad evaluation without task-specific feature engineering. Cells thus group programs with similar scores, and a lineage is redundant when its offspring revisit previously reached score levels without improvement. Richer descriptors can be used without modifying the method; we verify on two tasks that they perform comparably to the combined score (Appendix C.1).

Behavioral Redundancy Rate. We call a mutation *redundant* if its child p' lands in a cell already on the parent's ancestral lineage, $C_{b'} \in \mathcal{B}(p)$, without improving on the parent, $f(p') \leq f(p)$, with $C_{b'}$ taken when p' is generated. The behavioral redundancy rate is $\text{BRR} = T_{\text{red}}/T$, where T_{red} is the number of redundant mutations among the T mutations of a run. Since the baselines do not maintain cells or lineages, we run the same cell maintenance as EVADE (§3.5) alongside each baseline; these cells are only observed and never guide the search. They also yield the search trees analyzed in §4.3.

4.1 EXPERIMENTAL RESULTS

Mathematical Optimization Benchmarks Table 1 reports results on the six mathematical optimization tasks. With GLM-5.2, EVADE achieves the best mean on 5 of 6 tasks and the best or tied-best single-run value on 4. With DeepSeek-V4-Flash, it achieves the best mean and the best or tied-best single-run value on all 6. On circle packing, its best runs exceed the human reference in the square setting with GLM-5.2 (2.6360) and match AlphaEvolve (a closed-source system by Google DeepMind) in the rectangle setting with DeepSeek-V4-Flash (2.3658). The largest margin appears on Signal Processing with GLM-5.2, where EVADE reaches a mean of 0.7700 against 0.7124 for the strongest baseline; we examine this run in §C.2.1.

Table 1: Mathematical optimization benchmarks. We report Mean ± Std and Best (Max) objective values over three runs. Higher is better for all metrics. **Bold** denotes the best performing method per task within each backbone. Underline denotes the second-best. (N values: Square=26, Rect=21, Tri=11, Convex=13, Max=3).

	CirclePack (Sq)		CirclePack (Rect)		Heilbronn (Tri)		Heilbronn (Cvx)		MinMax Dist		Signal Proc	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Human	–	2.6340	–	2.3640	–	.0360	–	.0306	–	.2399	–	–
AlphaEvolve	–	2.6350	–	2.3658	–	.0365	–	.0309	–	.2398	–	–
Backbone: GLM-5.2												
GEPA	2.5666±.0436	2.6169	2.3539±.0072	2.3621	.0221±.0118	.0307	.0240±.0013	.0249	.2249±.0119	.2338	.6361±.0393	.6707
ShinkaEvolve	2.5552±.0477	2.6100	2.3239±.0654	2.3621	.0282±.0038	.0326	.0255±.0014	.0269	**.2399**±.0002	**.2401**	.7124±.0152	.7279
EvoX	2.5806±.0506	2.6152	2.3560±.0093	**2.3647**	.0280±.0034	.0301	.0203±.0046	.0256	.2381±.0032	.2400	.6687±.0084	.6770
AdaEvolve	2.6258±.0137	**2.6360**	2.3242±.0476	2.3555	.0202±.0136	**.0337**	.0257±.0031	.0278	.2304±.0166	**.2401**	.6965±.0104	.7085
EVADE	**2.6281**±.0092	**2.6360**	**2.3562**±.0018	2.3581	**.0309**±.0013	.0321	**.0269**±.0012	**.0280**	.2370±.0029	**.2401**	**.7700**±.0370	**.8120**
Backbone: DeepSeek-V4-Flash												
GEPA	2.5168±.0215	2.5414	2.3396±.0226	2.3547	.0217±.0036	.0242	.0226±.0011	.0239	.2248±.0119	.2337	.6540±.0367	.6834
ShinkaEvolve	2.4429±.0402	2.4879	2.2376±.0959	2.3479	.0267±.0020	.0279	.0247±.0008	.0256	.2091±.0055	.2131	.6592±.0042	.6619
EvoX	2.6130±.0116	2.6205	2.3504±.0047	2.3557	.0277±.0024	.0304	.0244±.0004	.0247	.2223±.0301	**.2401**	.6625±.0135	.6772
AdaEvolve	2.5790±.0359	2.6203	2.3542±.0077	2.3621	.0300±.0009	.0307	.0234±.0039	.0267	.2290±.0094	.2398	.7062±.0217	.7248
EVADE	**2.6239**±.0062	**2.6310**	**2.3624**±.0030	**2.3658**	**.0311**±.0027	**.0342**	**.0276**±.0003	**.0278**	**.2348**±.0091	**.2401**	**.7133**±.0291	**.7383**

Table 2: System optimization benchmarks. We report Mean ± Std and Best objective values over three runs. Arrows indicate optimization direction (↑: higher is better, ↓: lower is better). **Bold** denotes the best performing method per task within each backbone. Underline denotes the second-best. Human/SOTA reference values are shown where available.

	Cloudcast (↓)		LLM-SQL (↑)		Prism (↑)		EPLB (↑)		TXN (↑)	
Strategy	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best
Human / SOTA	–	626.20	–	.6920	–	21.89	–	.1265	–	2725.0
Backbone: GLM-5.2										
GEPA	837.49±91.25	756.06	.6843±.0217	.7048	26.32±0.45	26.79	.1293±.0001	.1294	4186.0±110.6	4310.3
ShinkaEvolve	652.33±20.22	630.13	.7060±.0026	.7086	26.42±0.15	26.54	.1298±.0007	.1306	4149.8±51.7	4201.7
EvoX	667.12±69.28	627.12	.7005±.0130	**.7155**	26.25±0.10	26.36	.1282±.0006	.1288	4024.6±138.0	4166.7
AdaEvolve	634.67±3.79	632.00	.5037±.3556	.7141	26.31±0.10	26.43	.1449±.0018	.1464	**4237.3**±192.5	**4366.8**
EVADE	**623.33**±9.24	**618.00**	**.7084**±.0046	.7135	**26.75**±0.36	**27.16**	**.1462**±.0006	**.1467**	4210.8±141.7	4310.4
Backbone: DeepSeek-V4-Flash										
GEPA	830.00±154.57	687.00	.7076±.0166	**.7267**	24.64±1.71	26.19	.1322±.0101	.1438	4055.3±95.8	4166.0
ShinkaEvolve	651.00±22.65	627.00	.6778±.0143	.6920	26.20±0.08	26.25	.1345±.0091	.1450	3747.0±102.4	3861.0
EvoX	751.52±154.79	657.12	.7047±.0094	.7120	26.28±0.10	26.36	.1266±.0005	.1270	4060.0±50.6	4115.2
AdaEvolve	668.00±22.11	651.00	.7073±.0187	.7240	26.34±0.14	26.50	.1382±.0096	.1439	3987.3±139.0	4098.4
EVADE	**621.00**±5.20	**618.00**	**.7152**±.0087	.7247	**26.50**±0.16	**26.65**	**.1404**±.0096	**.1467**	**4161.3**±53.1	**4219.4**

ADRS Systems Benchmark Table 2 reports results on the five ADRS systems tasks. With GLM-5.2, EVADE achieves the best mean on 4 out of 5 tasks and the best single-run value on 3 tasks. With DeepSeek-V4-Flash, it achieves the best mean on all 5 tasks and the best single-run value on 4 tasks. Among these tasks, Cloudcast is noteworthy: with both backbones, EVADE reaches a cost of 618.00 and is the only method whose mean outperforms the human SOTA of 626.20 (623.33 with GLM-5.2 and 621.00 with DeepSeek-V4-Flash). §C.2.2 traces how this solution is found. On LLM-SQL with GLM-5.2, AdaEvolve's mean drops to 0.5037 ± 0.3556 because one run stays in an early local optimum for the entire budget; EVADE escapes a similar optimum in its run (§C.2.3).

Behavioral Redundancy Rate Figure 2 shows the behavioral redundancy rate (BRR), i.e., how often a method regenerates behaviors that it has already produced. With GLM-5.2, EVADE has the lowest BRR on the mathematical tasks, with a median below 10% and a narrow spread, while the baselines mostly lie between roughly 25% and 60%. EVADE thus spends far fewer evaluations on behaviors it has already explored. The same conclusion holds for the full dataset and across LLM backbones, as shown in Figures 5b, 5c, 5a, and 5d.

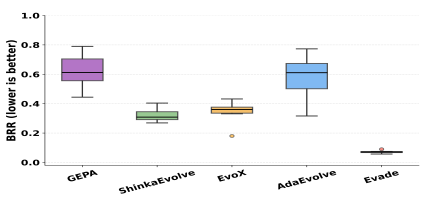


Figure 2: BRR (lower is better) on the mathematical tasks with GLM-5.2.

Table 3: Ablation study on Signal Processing and Prism: EVADE vs variants that replace individual or all components with competing methods. Results report mean ± std over $n = 3$ runs.

Components			Signal Processing		Prism	
EPS	EGM	EMS	Fitness (↑)	BRR % (↓)	Fitness (↑)	BRR % (↓)
			0.696 ± 0.018	13.3 ± 4.3	26.31 ± 0.10	17.7 ± 12.1
	✓	✓	0.708 ± 0.027	14.3 ± 4.0	26.34 ± 0.09	17.0 ± 3.6
✓		✓	0.719 ± 0.025	7.0 ± 5.2	26.35 ± 0.17	14.7 ± 3.2
✓	✓		0.726 ± 0.008	9.3 ± 5.7	26.47 ± 0.19	13.7 ± 7.6
✓	✓	✓	$\mathbf{0.770 \pm 0.037}$	$\mathbf{5.7 \pm 0.6}$	$\mathbf{26.75 \pm 0.36}$	$\mathbf{12.3 \pm 4.2}$

4.2 ABLATION STUDIES

We conduct ablation experiments with GLM-5.2 on Signal Processing from the mathematical benchmarks and Prism from ADRS. We isolate three core modules: Evolvability-guided Parent Selection (EPS), Evolvability-guided Mutation (EGM), and Evolvability-guided Mode Selection (EMS), and replace them with competing methods. Specifically, we replace EPS with the parent selection method based on fitness and code difference as in AdaEvolve and ShinkaEvolve. Here, we adopt the AdaEvolve implementation for code difference (see Appendix C for details). For EGM, we replace it with a variant that provides the full experience history at every step, without evolvability-based pruning. Finally, we disable EMS and sample each mode uniformly. Table 3 summarizes the results, showing that replacing any component lowers fitness and raises BRR.

Impact of Evolvability-Guided Parent Selection (EPS). Replacing EPS causes the largest drop in fitness on both tasks ($0.770 \rightarrow 0.708$ on Signal Processing and $26.75 \rightarrow 26.34$ on Prism) and the largest rise in BRR ($5.7\% \rightarrow 14.3\%$ and $12.3\% \rightarrow 17.0\%$). Without EPS, BRR returns to roughly the level of the variant without all three modules (13.3% and 17.7%), indicating that EPS accounts for most of the reduction in redundancy. *Selection based on fitness and code difference revisits previously explored behaviors, while evolvability-based selection helps avoid redundant mutations.*

Impact of Evolvability-Guided Mutation (EGM). Replacing EGM with the full experience history causes only a small change in BRR ($5.7\% \rightarrow 7.0\%$ on Signal Processing and $12.3\% \rightarrow 14.7\%$ on Prism), but substantially reduces fitness ($0.770 \rightarrow 0.719$ and $26.75 \rightarrow 26.35$). EGM thus primarily affects mutation quality rather than the frequency of revisiting old behaviors. We hypothesize that retaining outdated experience constrains the LLM to make small adjustments to previously explored ideas, causing the search to stagnate rather than generate novel ones.

Impact of Evolvability-Guided Mode Selection (EMS). We replace EMS with a module that samples modes uniformly. Fitness decreases on both tasks ($0.770 \rightarrow 0.726$ and $26.75 \rightarrow 26.47$), while BRR increases ($5.7\% \rightarrow 9.3\%$ and $12.3\% \rightarrow 13.7\%$), with smaller changes than those caused by removing EPS or EGM. On Prism, all ablated variants fall within 0.2 of one another (26.31–26.47) and exhibit lower variance than the full method, but none exceeds 26.5.

Search Trajectory Dynamics. Figure 3 shows the best run of each variant. EVADE starts slower: it trails the ablated variants for about the first 40% of the budget on Signal Processing. Most ablated variants rely more on signals that can be measured right away, such as fitness, so they improve quickly at first. They then plateau, because they keep mutating the directions that scored well early. Evolvability, in contrast, has to be learned from mutation outcomes, so EVADE first spends budget trying directions. Once it knows which cells can still improve, it moves there and reaches scores no ablated variant reaches: 0.812 on Signal Processing. The observation on Prism is similar, which is shown in the Appendix B.1.

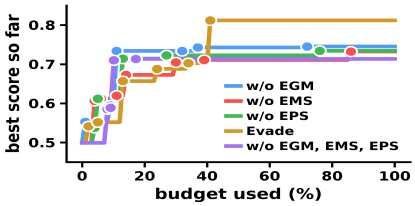


Figure 3: Optimization trajectories of EVADE and its ablated variants across iterations on Signal Processing task.

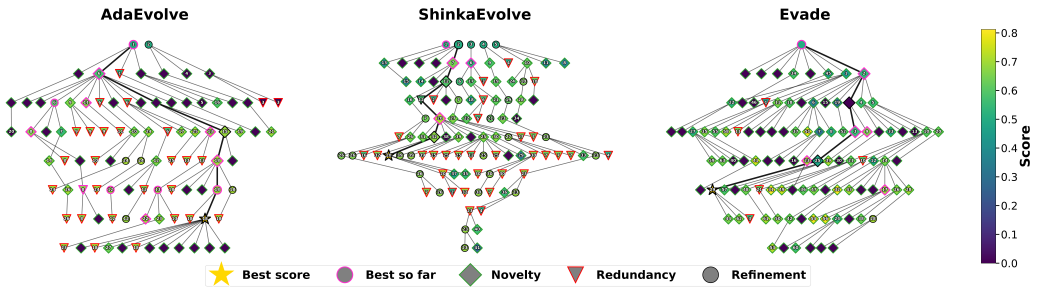


Figure 4: **Search trees of AdaEvolve, ShinkaEvolve, and EVADE** on Signal Processing. Each node is a behavioral cell, colored by fitness. Red triangles mark redundant mutations, whose child returns to a cell already on its lineage without improving on its parent; green diamonds mark mutations that reach a new cell. The star marks the best solution.

4.3 CASE STUDIES: REDUCING BEHAVIORAL REDUNDANCY ON SIGNAL PROCESSING

Figure 4 shows the search trees of AdaEvolve, ShinkaEvolve, and EVADE on Signal Processing. Both baselines spend much of their budget on redundant search (as shown as the dominating number of red triangles), concentrating their mutations around a few high-fitness cells, yet they fail for different reasons. AdaEvolve repeatedly samples high-scoring programs and regenerates existing methods without exploring novel directions. ShinkaEvolve enforces syntactic code diversity, but still produces functionally equivalent behaviors. AdaEvolve wastes search budget by repeatedly generating adaptive Holt filters and Savitzky–Golay variants; meanwhile, ShinkaEvolve creates one candidate that increases the Hodrick–Prescott (HP) filter penalty parameter while removing post-processing, and another that replaces a dense matrix solver with a sparse formulation—despite these implementation differences, both candidates yield identical output behaviors and fitness metrics. Consequently, BRR reaches 31.6% and 35.7% on Signal Processing for AdaEvolve and ShinkaEvolve, respectively, compared with only 5.7% for EVADE.

EVADE avoids wasting search budget by distributing its search across distinct lineages. On Signal Processing, it progresses from an adaptive Holt filter (0.542) through Savitzky–Golay variants (0.552, 0.657) to an L1 trend-filtering pipeline (0.688) and a spline trend-lock filter (0.703). At iteration 41, it selects a parent with fitness only 0.475 because its cell retains high evolvability. Mutating this parent produces a zero-phase Butterworth filter with the best score of 0.812.

5 DISCUSSION

Contributions. This work introduces EVADE, an LLM-guided evolutionary search framework driven by an evolvability signal computed from the outcomes of mutations the search already performs. The signal guides the balance between exploration and exploitation, parent selection, and the experience supplied to each mutation. Across 11 mathematical and systems optimization tasks, EVADE achieves the best mean on most tasks with two backbones while substantially reducing redundant search, and discovers an algorithm that surpasses the human state of the art on Cloudcast. By releasing our code, we hope to support further study of evolvability in LLM-guided search.

Limitations. Our experiments use the combined score as the behavior descriptor, and we evaluate richer descriptors on only two tasks. We also lack an analysis of how the evolvability signal evolves over the course of the search, and performance depends on the hyperparameters of the signal. Finally, as in prior work, we report three runs per setting because each run is costly and time-consuming.

Extensions. Adapting the signal's hyperparameters during the search would reduce manual tuning, and domain-specific behavior descriptors could separate programs that reach similar scores through different mechanisms. The signal could also guide methods that train the generator for a specific discovery task. Finally, extending the search to systems of many interacting modules, with evolvability estimated per module, is a step toward discovering solutions that are useful in practice.

AI USE STATEMENT

We used generative AI tools in two ways. First, to aid and polish the writing of this paper, including rephrasing sentences and checking grammar and consistency of notation. Second, to assist with implementing the code of EVADE and the experimental pipeline; all AI-assisted code was reviewed, tested, and verified by the authors. The conceptualization, algorithmic design, experimental design, and interpretation of the results were carried out by the authors. We take full responsibility for the final content of this work, including all text, code, and claims. Separately, large language models are a component of the method under study: they serve as the mutation operator in EVADE and in all baselines, as described in §4.

ETHICS STATEMENT

This work does not involve human subjects, personal data, or newly collected datasets. All experiments use publicly available benchmarks from the SkyDiscover suite (Liu et al., 2026b). Programs generated by the language models are executed only by the benchmark evaluators. We do not foresee direct negative societal impacts beyond those of automated algorithm discovery in general, whose outputs should be verified before deployment.

REPRODUCIBILITY STATEMENT

The source code of EVADE, including the configurations used in our experiments, is available at the anonymous repository linked in the abstract. Section 4 specifies the tasks, baselines, backbone models, budget, and number of runs. The complete procedure is given in Algorithm 1 and its subroutines in the appendix. The appendix also lists the mode-specific weights and inspiration rules (Table 4), the prompts, the hyperparameters of the evolvability signal together with a sensitivity analysis (Table 5), and the computation of code difference used in the ablation (Appendix C).

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Lee Altenberg et al. The evolution of evolvability in genetic programming. *Advances in genetic programming*, 3:47–74, 1994.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. 2025. URL https://arxiv.org/abs/2510.14150.

Roberto Battiti and Giampietro Tecchiolli. The reactive tabu search. *ORSA journal on computing*, 6 (2):126–140, 1994.

Erick Cantu-Paz. *Efficient and accurate parallel genetic algorithms*, volume 1. Springer Science & Business Media, 2000.

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, 2026. URL https://arxiv.org/abs/2602.20133.

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, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. Barbarians at the gate: How ai is upending systems research, 2025b. URL https://arxiv.org/abs/2510.06189.

Antoine Cully and Yiannis Demiris. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 22(2):245–259, 2017.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence, 2026.

Stephane Doncieux, Giuseppe Paolo, Alban Laflaquière, and Alexandre Coninx. Novelty search makes evolvability inevitable. In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference*, pp. 85–93, 2020.

Andrea Ferigo, LB Soros, Eric Medvet, and Giovanni Iacca. On the entanglement between evolvability and fitness: An experimental study on voxel-based soft robots. In *Artificial Life Conference Proceedings 34*, volume 2022, pp. 15. MIT press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2022.

Alexander Gajewski, Jeff Clune, Kenneth O Stanley, and Joel Lehman. Evolvability es: Scalable and direct optimization of evolvability. In *Proceedings of the genetic and evolutionary computation conference*, pp. 107–115, 2019.

Bruno Gasperov and Marko Durasevic. On evolvability and behavior landscapes in neuroevolutionary divergent search. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pp. 1203–1211, 2023.

GLM-5-Team, :, Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, Chenzheng Zhu, Congfeng Yin, Cunxiang Wang, Gengzheng Pan, Hao Zeng, Haoke Zhang, Haoran Wang, Huilong Chen, Jiajie Zhang, Jian Jiao, Jiaqi Guo, Jingsen Wang, Jingzhao Du, Jinzhu Wu, Kedong Wang, Lei Li, Lin Fan, Lucen Zhong, Mingdao Liu, Mingming Zhao, Pengfan Du, Qian Dong, Rui Lu, Shuang-Li, Shulin Cao, Song Liu, Ting Jiang, Xiaodong Chen, Xiaohan Zhang, Xuancheng Huang, Xuezhen Dong, Yabo Xu, Yao Wei, Yifan An, Yilin Niu, Yitong Zhu, Yuanhao Wen, Yukuo Cen, Yushi Bai, Zhongpei Qiao, Zihan Wang, Zikang Wang, Zilin Zhu, Ziqiang Liu, Zixuan Li, Bojie Wang, Bosi Wen, Can Huang, Changpeng Cai, Chao Yu, Chen Li, Chengwei Hu, Chenhui Zhang, Dan Zhang, Daoyan Lin, Dayong Yang, Di Wang, Ding Ai, Erle Zhu, Fangzhou Yi, Feiyu Chen, Guohong Wen, Hailong Sun, Haisha Zhao, Haiyi Hu, Hanchen Zhang, Hanrui Liu, Hanyu Zhang, Hao Peng, Hao Tai, Haobo Zhang, He Liu, Hongwei Wang, Hongxi Yan, Hongyu Ge, Huan Liu, Huanpeng Chu, Jia'ni Zhao, Jiachen Wang, Jiajing Zhao, Jiamin Ren, Jiapeng Wang, Jiaxin Zhang, Jiayi Gui, Jiayue Zhao, Jijie Li, Jing An, Jing Li, Jingwei Yuan, Jinhua Du, Jinxin Liu, Junkai Zhi, Junwen Duan, Kaiyue Zhou, Kangjian Wei, Ke Wang, Keyun Luo, Laiqiang Zhang, Leigang Sha, Liang Xu, Lindong Wu, Lintao Ding, Lu Chen, Minghao Li, Nianyi Lin, Pan Ta, Qiang Zou, Rongjun Song, Ruiqi Yang, Shangqing Tu, Shangtong Yang, Shaoxiang Wu, Shengyan Zhang, Shijie Li, Shuang Li, Shuyi Fan, Wei Qin, Wei Tian, Weining Zhang, Wenbo Yu, Wenjie Liang, Xiang Kuang, Xiangmeng Cheng, Xiangyang Li, Xiaoquan Yan, Xiaowei Hu, Xiaoying Ling, Xing Fan, Xingye Xia, Xinyuan Zhang, Xinze Zhang, Xirui Pan, Xu Zou, Xunkai Zhang, Yadi Liu, Yandong Wu, Yanfu Li, Yidong Wang, Yifan Zhu, Yijun Tan, Yilin Zhou, Yiming Pan, Ying Zhang, Yinpei Su, Yipeng Geng, Yong Yan, Yonglin Tan, Yuean Bi, Yuhan Shen, Yuhao Yang, Yujiang Li, Yunan Liu, Yunqing Wang, Yuntao Li, Yurong Wu, Yutao Zhang, Yuxi Duan, Yuxuan Zhang, Zezhen Liu, Zhengtao Jiang, Zhenhe Yan, Zheyu Zhang, Zhixiang Wei, Zhuo Chen, Zhuoer Feng, Zijun Yao, Ziwei Chai, Ziyuan Wang, Zuzhou Zhang, Bin Xu, Minlie Huang, Hongning Wang, Juanzi Li, Yuxiao Dong, and Jie Tang. Glm-5: from vibe coding to agentic engineering, 2026. URL https://arxiv.org/abs/2602.15763.

Qinglong Hu and Qingfu Zhang. Partition to evolve: Niching-enhanced evolution with LLMs for automated algorithm discovery. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL https://openreview.net/forum?id=OEawM2coNT.

Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=ownPRHrXMx.

Adam Katona, Daniel W Franks, and James Alfred Walker. Quality evolvability es: Evolving individuals with a distribution of well performing and diverse offspring. In *Artificial Life Conference Proceedings 33*, volume 2021, pp. 81. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2021.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=lKEdGCoDNC.

Joel Lehman and Kenneth O Stanley. Evolvability is inevitable: Increasing evolvability without the pressure to adapt. *PloS one*, 8(4):e62186, 2013.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pp. 331–366. Springer, 2023.

Fei Liu, Tong Xialiang, 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 *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=BwAkaxqiLB.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. Evox: Meta-evolution for automated discovery, 2026a. URL https://arxiv.org/abs/2602.23413.

Shu Liu, Mert Cemri, Shubham Agarwal, Alexander Krentsel, Ashwin Naren, Qiuyang Mang, Zhifei Li, Akshat Gupta, Monishwaran Maheswaran, Audrey Cheng, Melissa Pan, Ethan Boneh, Kannan Ramchandran, Koushik Sen, Matei Zaharia, Alexandros G. Dimakis, and Ion Stoica. Skydiscover: A flexible, adaptive framework for ai-driven scientific and algorithmic discovery. In *Proceedings of the ACM Conference on AI and Agentic Systems*, CAIS '26, pp. 1223–1227. Association for Computing Machinery, 2026b. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, 1976. doi: 10.1109/TSE.1976.233837.

Henok Mengistu, Joel Lehman, and Jeff Clune. Evolvability search: directly selecting for evolvability in order to study and produce it. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, pp. 141–148, 2016.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015. URL https://arxiv.org/abs/1504.04909.

Glenford J. Myers. An extension to the cyclomatic measure of program complexity. *SIGPLAN Not.*, 12(10):61–64, October 1977. ISSN 0362-1340. doi: 10.1145/954627.954633. URL https://doi.org/10.1145/954627.954633.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2023. doi: 10.1038/s41586-023-06924-6.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *The Thirteenth International Conference on Learning Representations*, 2025.

Rich Sutton. The bitter lesson, 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Günter P Wagner and Lee Altenberg. Perspective: complex adaptations and the evolution of evolvability. *Evolution*, 50(3):967–976, 1996.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and yelong shen. Thetaevolve: Test-time learning on open problems. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=XHP568OuIl.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=483IPG0HWL.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=96zNuQrH9Y.

A Extension of Evade Framework Section

A.1 Algorithm Summary

Algorithm 1 Evade – Main Loop. Cells, lineage and experience are maintained so that evolvability can be estimated over them; the estimate then guides selection and mutation on the following iteration.

Require: Initial program p_0, evaluator F, budget T, generator model M, smoothing constant κ

$A \leftarrow \{p_0\}, \quad \mathcal{C} \leftarrow \{C_1 = \{p_0\}\}, \quad \mathcal{B}(p_0) \leftarrow \{C_1\}, \quad \mathcal{E} \leftarrow \emptyset$

$\bar{e} \leftarrow 1/2, \quad e(C_1) \leftarrow \bar{e}$

for $t = 1$ **to** T **do**

// Evolvability-guided selection

$k \leftarrow \textsc{SelectMode}(\bar{e})$ ▷ Eq. 10

$C_b \leftarrow \textsc{SelectCell}(\mathcal{C}, k)$ ▷ Eq. 7

$p \leftarrow \textsc{SelectParent}(C_b)$ ▷ Alg. 3

$Z \leftarrow \textsc{SelectInspires}(p, k)$ ▷ Alg. 4

// Evolvability-guided mutation

$E \leftarrow \textsc{RegulateExp}\big(\mathcal{E}, C_b, e(C_b)\big)$ ▷ Eq. 8

$g' \sim \pi_k^g(\cdot \mid p, E, Z)$

$p' \sim \pi_k^c(\cdot \mid p, g', E, Z)$ ▷ Eq. 9

// Evaluation

$f' \leftarrow F(p'), \quad \beta' \leftarrow$ behavior descriptor of p'

$A \leftarrow A \cup \{p'\}$

// Evolvability estimation

$C_{b'} \leftarrow \textsc{AssignCell}(\beta', \mathcal{C})$ ▷ Alg. 5

$\mathcal{B}(p') \leftarrow \mathcal{B}(p) \cup \{C_{b'}\}$

$r_t(p) \leftarrow \textsc{EvolvabilitySignal}\big(p, p', C_{b'}, \mathcal{B}(p), e(C_{b'})\big)$ ▷ Eq. 1

$(S_C, N_C) \leftarrow \textsc{UpdateRecords}(p, r, \mathcal{C})$ ▷ Eq. 4, **??**

$e(C) \leftarrow \dfrac{S_C + \kappa\,\bar{e}}{N_C + \kappa}$ for all $C \in \mathcal{C}$ ▷ cell level, Eq. 5

$\bar{e} \leftarrow \dfrac{\sum_C N_C\, e(C)}{\sum_C N_C}$ ▷ global level, Eq. 6

// Structure maintenance

$\mathcal{C}, \mathcal{B}, \{e(C)\}, \bar{e} \leftarrow \textsc{Recluster}(A, \mathcal{C})$ ▷ Alg. 6

$\mathcal{E} \leftarrow \textsc{UpdateExperience}(\mathcal{E}, p, p', g, C_b, t)$ ▷ Alg. 7

end for

return $\arg\max_{p \in A} F(p)$

A.2 Extension of Evolvability-Guided Selection

Mode Selection. We compute a mode selection probability $\Pr(k)$ for each mode k, and draw one mode at each iteration accordingly:

$$\Pr(k) = \frac{\exp\big(-|z_t - \mu_k|/\eta\big)}{\sum_j \exp\big(-|z_t - \mu_j|/\eta\big)}, \tag{10}$$

where $z_t \in [0, 1]$ estimates the productivity of the search over its recent steps, derived from the global evolvability $\bar{e}_t$ (Algorithm 2), and $\mu_k \in \{0, \frac{1}{3}, \frac{2}{3}, 1\}$ are fixed, evenly spaced anchors, one per mode: breakthrough at 0, explore at $\frac{1}{3}$, balance at $\frac{2}{3}$, and exploit at 1. A high z_t favors exploit, and a low z_t favors breakthrough. The temperature η controls how sharply $\Pr(k)$ concentrates on the closest mode. We set $\eta = 0.2$, so that the mode closest to z_t dominates while adjacent modes remain reachable. Sampling a mode this way keeps every mode reachable at every point in the search.

Algorithm 2 computes z_t from a sliding window of recent values of $\bar{e}_t$. Because the number of observed cells grows over the run, $\bar{e}_t$ tends to decrease even when the search performs as well as before. Ranking the raw values would therefore push z_t toward 0 and the draw toward breakthrough. We instead remove the linear trend of the window and rank the most recent residual, so that z_t rises or falls only when the search does better or worse than its recent trajectory predicts.

Algorithm 2 Computation of z_t. OBSERVE is called once per iteration; COMPUTEZ is called whenever a mode is drawn.

Require: window size W, minimum fill $W_{\min}$, warm-up boundary w; sliding window H of past global evolvability values (capacity W), initially empty

function OBSERVE($\bar{e}_t,\ t$)
  if $t \geq w$: append $\bar{e}_t$ to H, evicting the oldest entry once $|H| > W$
end function

function COMPUTEZ( )
  if $|H| < W_{\min}$ **return** $\frac{1}{2}$ ▷ insufficient history
  $(a, b) \leftarrow \arg\min_{a,\,b} \sum_{i=0}^{|H|-1} \big(H_i - a - b\,i\big)^2$ ▷ linear trend of the window
  $r_i \leftarrow H_i - (a + b\,i), \quad i = 0, \ldots, |H| - 1$ ▷ detrended values
  $v \leftarrow r_{|H|-1}$ ▷ most recent residual
  return $\dfrac{\big|\{\, i : r_i < v \,\}\big| + \frac{1}{2}\big|\{\, i : r_i = v \,\}\big|}{|H|}$ ▷ mid-rank of v in the window
end function

Cell Selection: Mode-Specific Weights. Table 4 lists, for each mode, the weights of fitness and evolvability in the cell utility (Eq. equation 7), the source of the $|\mathcal{I}_t^{(k)}| = 4$ inspiration programs, and the prompt used. The weights are fixed across all tasks and backbones.

Table 4: Cell-selection weights, inspiration programs, and prompt of each mode.

Mode	$w_f^{(k)}$	$w_e^{(k)}$	Inspiration programs	Prompt
Exploit	0.8	0.2	programs in the parent's cell	Exploitation
Balance	0.5	0.5	programs in the parent's cell	Exploitation
Explore	0.4	0.6	best program of each other cell	Exploration
Breakthrough	0.7	0.3	best program of each other cell	Exploration

The weights determine which cells are mutated. Exploit weights fitness most heavily to refine strong cells, but keeps a small weight on evolvability so that a strong cell whose mutations no longer succeed is not sampled indefinitely. Balance weights the two equally. Explore favors evolvability, directing the budget to cells whose mutations still reach new behaviors or improvements. Breakthrough explores through its inspiration programs rather than through cell selection: it combines ideas from many cells, which requires a strong base program. AdaEvolve, for the same reason, builds its breakthrough tactics from the global best program (Cemri et al., 2026). We likewise weight fitness heavily, but keep a weight of 0.3 on evolvability, since the best program may itself be exhausted; always building on it can trap the search, as in the LLM-SQL case study (§C.2.3).

Parent & Inspiration Program Selection: Program Complexity. Within the drawn cell C_b, the parent is selected by structural simplicity. We measure the complexity of a program p with a lightweight surrogate for the cyclomatic complexity of McCabe (1976), the number of linearly independent paths through the control-flow graph, computed as the number of decision predicates plus one. We approximate the number of predicates by counting the branch- and loop-introducing keywords in the source text, together with the short-circuit operators `and` and `or`, so that a condition with k clauses contributes k rather than 1, following the standard extension of the measure to compound predicates (Myers, 1977):

$$c(p) = 1 + \sum_{w \in \mathcal{W}} \mathrm{occ}(p, w), \qquad \mathcal{W} = \{\texttt{if}, \texttt{elif}, \texttt{for}, \texttt{while}, \texttt{except}, \texttt{and}, \texttt{or}\}, \tag{11}$$

where $\mathrm{occ}(p, w)$ is the number of occurrences of keyword w in the source of p. The count also includes keywords inside comments and string literals, which is acceptable because $c(p)$ is only used to rank programs within the same cell. The complexity is min–max normalized within the cell,

and the parent is drawn with probability

$$\Pr(p \mid C_b) = \frac{1 - \hat{c}(p)}{\sum_{q \in C_b} (1 - \hat{c}(q))}, \qquad \hat{c}(p) = \frac{c(p) - \min_{q \in C_b} c(q)}{\max_{q \in C_b} c(q) - \min_{q \in C_b} c(q)}, \tag{12}$$

with a uniform draw when all members of C_b have equal complexity (Algorithm 3).

Algorithm 3 SELECTPARENT: draw a parent within a cell by structural simplicity.

function SELECTPARENT(C_b)
  if $|C_b| = 1$ **then**
    return the single member of C_b
  end if
  compute $c(q)$ for all $q \in C_b$ ▷ Eq. equation 11
  if $\max_q c(q) = \min_q c(q)$ **then**
    return p drawn uniformly from C_b
  end if
  return p drawn with probability $\Pr(p \mid C_b)$ ▷ Eq. equation 12
end function

Parent & Inspiration Program Selection: Inspiration Programs of Each Mode. Island-based systems draw inspiration programs from the parent's own subpopulation to refine it, and from the best programs elsewhere to introduce different ideas (Romera-Paredes et al., 2023; Sharma, 2025; Cemri et al., 2026). We follow the same principle with behavioral cells. Exploit and balance draw the highest-scoring programs in the parent's cell, contrasting variants of the same behavior so that the generator can refine it. Explore and breakthrough draw the best program of each other cell, contrasting different behaviors so that the generator can move toward or combine them. If fewer than four programs are available, the set is completed with the highest-scoring programs of the archive (Algorithm 4).

Algorithm 4 SELECTINSPIRES: select the inspiration programs according to the mode.

function SELECTINSPIRES($p \in C_b,\ k$)
  if $k \in \{\text{EXPLORE}, \text{BREAKTHROUGH}\}$ **then**
    $\mathcal{I} \leftarrow$ highest-scoring program of each cell $C \neq C_b$
  else
    $\mathcal{I} \leftarrow$ highest-scoring programs in $C_b \setminus \{p\}$
  end if
  complete $\mathcal{I}$ with the highest-scoring programs of A_t if $|\mathcal{I}| < 4$
  return the four highest-scoring programs of $\mathcal{I}$
end function

A.3 EXTENSION OF EVOLVABILITY-GUIDED MUTATION

Mutation Operator: Prompts of Each Mode. Exploit and balance use the exploitation prompt; explore and breakthrough use the exploration prompt.

Mode context — Exploitation

PARENT SELECTION CONTEXT
This parent was selected from a high-performing region of the archive.

OPTIMIZATION GUIDANCE

- The solution already works well, but it can still be improved.
- Either refine the current approach or replace it with a better algorithm.
- Consider algorithmic changes, better data structures, and more efficient libraries.
- Preserve correctness.

Your goal: improve upon this solution.

Algorithm 5 ASSIGNCELL – assign a newly evaluated program to a behavioral cell. The program joins the closest existing cell unless its behavior descriptor lies farther than τ from every cell center, in which case it starts a new cell. τ is not a fixed constant: RECLUSTER (Alg. 6) resets it each time from the spread of behaviors currently observed, via NOVELTYRADIUS. Descriptors are compared through ϕ, a per-dimension robust rescaling that puts channels with different units on the same footing; before the first RECLUSTER, τ is undefined and every program forms its own cell.

function ASSIGNCELL(β', $\mathcal{C}$, τ)
  $C^\star \leftarrow$ the cell in $\mathcal{C}$ whose center is closest to β' under ϕ
  $d^\star \leftarrow$ that distance, $\|\phi(\beta') - \mu_{C^\star}\|$
  if $d^\star \le \tau$ **then**
    return $(C^\star,$ EXISTING$)$ ▷ same behavior as a cell already seen
  else
    create a new cell C_{new} with center $\mu_{C_{\text{new}}} \leftarrow \phi(\beta')$
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{C_{\text{new}}\}$
    return $(C_{\text{new}},$ NEW$)$ ▷ a behavior no existing cell covers
  end if
end function

function NOVELTYRADIUS(A) ▷ called by RECLUSTER; A is every evaluated program
  $D \leftarrow$ the distances between all pairs of *distinct* descriptors in A, under ϕ
  $\tau \leftarrow$ the 25th percentile of D ▷ a typical small gap between behaviors, not a fixed constant
  return τ
end function

Mode context — Exploration

PARENT SELECTION CONTEXT
This parent was selected to explore a different region of the search space.

EXPLORATION GUIDANCE

- Consider alternative algorithmic approaches.
- Do not be bound by the parent's current approach.
- Look for fundamentally different algorithms or new techniques.
- Keep new ideas correct.

Your goal: find new approaches that may outperform the current solutions.

A.4 EXTENSION OF MAINTENANCE

To reliably estimate evolvability and provide grounded feedback for LLM mutations, EVADE continuously maintains two components: the behavioral cell partition and the experience bank.

Behavioral Cell Maintenance. The behavioral space is organized into discrete cells $\mathcal{C}_t$ so that programs with similar behavior share a cell, while functionally distinct behaviors form separate cells. Whenever a new program p' is evaluated, ASSIGNCELL (Alg. 5) measures its distance to existing cell. If this distance is within an merge distance threshold ϵ, p' joins that cell; otherwise, it forms a new cell, distinguishing novel behaviors from existing ones. The ancestral lineage $\mathcal{B}(p)$ determines whether an offspring reaches an unexplored behavioral frontier ($C_{b'} \notin \mathcal{B}(p)$) or merely regresses to a cell its lineage has already occupied ($C_{b'} \in \mathcal{B}(p)$), directly governing reward assignment in Eq. equation 1.

To prevent the partition from depending on evaluation order, EVADE periodically rebuilds all cells via RECLUSTER (Alg. 6) every τ_{cell} iterations. In contrast to code-partitioning strategies (Hu & Zhang, 2025), we group programs by runtime behavior rather than static code semantics. Reclustering normalizes descriptors across dimensions, derives the distance threshold ϵ from the 25th percentile of pairwise distances, clusters behaviors via agglomerative grouping with cutoff ϵ, re-derives the ancestral lineages from the stored parent pointers, and updates the evolvability scores $e_t(C_b)$ via Eq. equation 5.

Algorithm 6 RECLUSTER – periodically rebuild the behavioral partition from scratch. Run for the first time at $t = w$ and every K_{cell} iterations after; a no-op otherwise. It fits over A, every evaluated program, not the surviving pool: a partition fitted on survivors would place τ using a sample already biased toward high scores, and a cell's records would leave the aggregate whenever the pool evicted their program. Recomputing the whole partition (rather than assigning incrementally) keeps it from being determined by the order behaviors were encountered. Cell ids are durable across rebuilds — an id names a region, not a slot — so the per-cell lateral bookkeeping and local experience survive a refit.

function RECLUSTER(A, $\mathcal{C}$)
  if the partition is fitted **and** $t \not\equiv w \pmod{K_{\text{cell}}}$ **return** $\big(\mathcal{C},\ \mathcal{B},\ \{e(C)\},\ \bar{e}\big)$
  $\phi \leftarrow \Phi(A)$ ▷ per-dimension median/IQR scaling over $\{\beta(p)\}_{p\in A}$; identity for hash / embedding descriptors
  $\tau \leftarrow$ NOVELTYRADIUS(A) ▷ Alg. 5
  $\mathcal{P} \leftarrow$ CAP$\big($ AGGLOMERATE$\big(\{\phi(\beta(p))\}_{p\in A},\ \tau\big),\ M_{\max}\ \big)$
  $\mathcal{C} \leftarrow$ RELABEL($\mathcal{P}$, $\mathcal{C}$)
  $C(p) \leftarrow$ the group of $\mathcal{C}$ containing p, for all $p \in A$
  $\mathcal{B}(p) \leftarrow \big\{\, C(q) \ :\ q \preceq p \,\big\}$, for all $p \in A$ ▷ cells on p's lineage, including its own; recomputed under the new labels
  $e(C) \leftarrow$ Eq. 5 for all $C \in \mathcal{C}$; $\bar{e} \leftarrow$ Eq. 6 ▷ same accumulated records (S_p, N_p), new cell membership
  return $\big(\mathcal{C},\ \mathcal{B},\ \{e(C)\},\ \bar{e}\big)$
end function

Algorithm 7 UPDATEEXPERIENCE. The store $\mathcal{E} = (G^+, G^-, \{L_b\})$ holds two global pools of statements about edits that improved (G^+) and degraded (G^-) fitness, and one pool L_b of strategies already tried from each cell C_b. R_t is the set of parent–child transitions since the previous global update; $\|$ concatenates, newest first. The policy $\pi_{\text{distil}}(\cdot \mid q, q')$ samples one neutral statement of what the edit $q \to q'$ did, conditioned on its diff and signed score change $\Delta(q, q') = \big(f(q') - f(q)\big)/|f_t^\star|$.

function UPDATEEXPERIENCE($\mathcal{E}$, p, p', g, C_b, t)
  $L_b \leftarrow$ UPDATELOCALEXPERIENCE$\big(L_b,\ g,\ p,\ p',\ t\big)$
  if $t \equiv 0 \pmod{\tau_{\text{exp}}}$ **then**
    $(G^+, G^-) \leftarrow$ UPDATEGLOBALEXPERIENCE$(G^+, G^-,\ R_t)$
  end if
  return $\mathcal{E} = \big(G^+, G^-,\ \{L_b\}\big)$
end function

function UPDATELOCALEXPERIENCE(L_b, g, p, p', t)
  if $g = \emptyset$ **return** L_b ▷ no strategy stated or inferable from the diff
  return $\langle\, g,\ \Delta(p, p'),\ t \,\rangle \,\|\, L_b$
end function

function UPDATEGLOBALEXPERIENCE(G^+, G^-, R)
  $S^+ \leftarrow \big(\ s \sim \pi_{\text{distil}}(\cdot \mid q, q')\ :\ (q \to q') \in R,\ \ f(q') > f(q)\ \big)$
  $S^- \leftarrow \big(\ s \sim \pi_{\text{distil}}(\cdot \mid q, q')\ :\ (q \to q') \in R,\ \ f(q') < f(q)\ \big)$
  return $\big(\ S^+ \| G^+,\ \ S^- \| G^-\ \big)$
end function

Experience Maintenance. Accumulating qualitative feedback is common in evolutionary LLM search (Ye et al., 2024; Lange et al., 2026; Cemri et al., 2026; Jiang et al., 2026). EVADE maintains it at two levels (Alg. 7). Local experience records, for each cell, the strategies already tried from it and their score changes, so that the LLM avoids repeating them. Global experience is updated every τ_{exp} iterations: an LLM describes what each recent edit did, and these descriptions are stored separately for edits that improved and edits that degraded fitness.

Experience Distillation Prompts. The policy π_{distil} in UPDATEGLOBALEXPERIENCE (Algorithm 7) is one LLM call per transition $q \to q'$ in R_t with $f(q') \neq f(q)$. The system prompt depends on the sign of the score change: edits that improved fitness are described for G^+, edits that degraded it for G^-. The user prompt is shared. It gives the two scores, the relative change with respect to the parent and to the global best (the signed change $\Delta(q, q')$ of Algorithm 7), the change description stated by the generator, and the code diff (truncated to 100 lines). Fields in braces are filled in at run time.

Distillation system prompt — Improvement (G^+)

You analyze one step of an automated algorithm-evolution experiment, where a child solution scored higher than its parent. State, as a neutral observation, the specific algorithmic change that is associated with this improvement. Be concise (2–4 sentences) and factual. Describe an association, not a rule: do not use prescriptive language such as 'must', 'should', 'always', or 'never'.

Distillation system prompt — Regression (G^-)

You analyze one step of an automated algorithm-evolution experiment, where a child solution scored lower than its parent. State, as a neutral observation, the specific algorithmic change that is associated with this regression. Be concise (2–4 sentences) and factual, without blame. Describe an association, not a rule: do not use prescriptive language such as 'must', 'should', 'always', or 'never'.

Distillation user prompt

PARENT score = {`parent_score`}
CHILD score = {`child_score`} ({`signed_pct`}% vs parent, {`signed_norm`}% vs global best)

CHANGES MADE
{`changes`}

CODE DIFF (parent → child)
{`diff`}

In 2–4 sentences, as a neutral observation: which algorithmic change is associated with this {`outcome`}?

B EXTENSION OF EXPERIMENTS

B.1 EXTENSION OF EXPERIMENTAL RESULTS

BRR Rate across tasks Figure 5 shows the behavioral redundancy rate (BRR), i.e., how often a method regenerates behaviors that it has already produced. With GLM-5.2, EVADE has the lowest BRR on the mathematical tasks, with a median below 10% and a narrow spread, while the baselines mostly lie between roughly 25% and 60%. EVADE thus spends far fewer evaluations on behaviors it has already explored. The same conclusion holds for the full dataset and across LLM backbones, as shown in Figures 5b, 5c, 5a, and 5d.

Extension of Ablation Studies

C CODE DIFFERENCE

We measure how much a program's code differs from the rest of the archive following the implementation of AdaEvolve (Liu et al., 2026b). The distance between two programs depends only on their source text, not on their behavior. It combines three terms: the Jaccard distance between their sets of tokens (identifiers and numeric literals of length at least two), the Jaccard distance between their sets of structural features (imports, function and class names, attribute calls of the form `a.b(...)`, and flags for `for`, `while`, `try`, `with`, `lambda`, `async`, and `yield`), and their relative difference in

length. The code difference of a program is its mean distance to its $k = 5$ nearest neighbors in the archive (Alg. 8).

Algorithm 8 CODEDIFFERENCE. $J(X, Y) = 1 - |X \cap Y|/|X \cup Y|$ denotes the Jaccard distance, $\mathrm{tok}(p)$ the token set, $\mathrm{feat}(p)$ the structural feature set, and $|p|$ the code length of program p.

function CODEDISTANCE($p,\ q$)
  $d_{\mathrm{tok}} \leftarrow J\big(\mathrm{tok}(p),\ \mathrm{tok}(q)\big)$
  $d_{\mathrm{feat}} \leftarrow J\big(\mathrm{feat}(p),\ \mathrm{feat}(q)\big)$
  $d_{\mathrm{len}} \leftarrow \big|\,|p| - |q|\,\big| \,/\, \max(|p|, |q|)$
  return $0.5\, d_{\mathrm{tok}} + 0.3\, d_{\mathrm{feat}} + 0.2\, d_{\mathrm{len}}$
end function

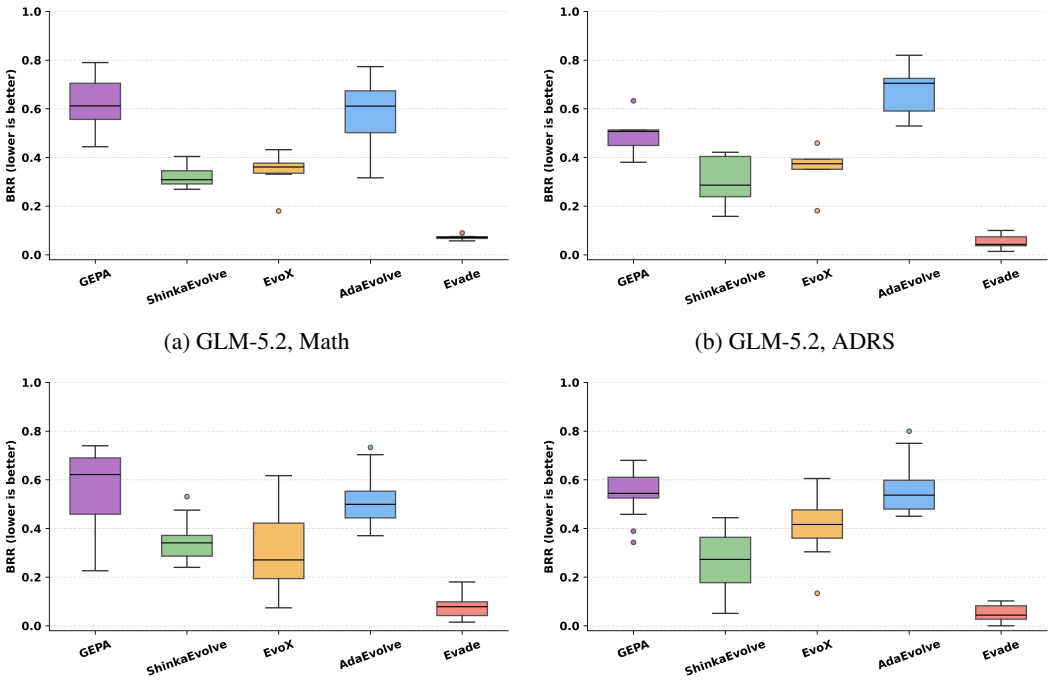
function CODEDIFFERENCE($p,\ A,\ k$)
  if $A \setminus \{p\} = \emptyset$ **then**
    return 1
  end if
  $D \leftarrow \big\{\, \textsc{CodeDistance}(p, q) : q \in A \setminus \{p\} \,\big\}$
  return mean of the $\min(k, |D|)$ smallest values in D
end function

/textbfResults. Figure 6 shows the best run of each variant. EVADE starts slower: it trails the ablated variants for about the first 40% of the budget on Signal Processing and about 70% on Prism. The ablated variants select parents by fitness and code difference, which can be measured right away, so they improve quickly at first. They then plateau, because they keep mutating the directions that scored well early. Evolvability, in contrast, has to be learned from mutation outcomes, so EVADE first spends budget trying directions. Once it knows which cells can still improve, it moves there and reaches scores no ablated variant reaches: 0.812 on Signal Processing and 27.16 on Prism.

C.1 HYPERPARAMETER SENSITIVITY AND BEHAVIOR DESCRIPTOR

Hyperparameter Sensitivity. We vary the decay factor ρ, the smoothing constant κ, and the floor α of the evolvability signal one at a time, keeping the others at their default values ($\rho = 0.9$, $\kappa = 4$, $\alpha = 0.5$), on Prism and Signal Processing with DeepSeek-V4-Flash (Table 5). The default configuration performs best on both tasks. The decay factor is the least sensitive for large values: $\rho \in \{0.75, 0.9, 0.95\}$ yield similar results on Prism, and $\rho = 0.95$ is close to the default on Signal Processing, while smaller values ($\rho \leq 0.6$) forget past outcomes too quickly and perform worse. The smoothing constant and the floor have a larger effect, and performance drops on both sides of the default. A small κ lets a cell's score change sharply after a few attempts, while a large κ keeps all cells close to the global prior and weakens the distinction between them. A small α gives little credit for reaching a new cell without improvement, while a large α rewards such moves almost as much as improvements. Most non-default settings remain within the range of the baselines except AdaEvolve, so the margin over the strongest baseline depends on the default configuration.

Behavior Descriptor. The main experiments use the combined score as the behavior descriptor (§4). To compare it with a richer descriptor, we replace it with a vector descriptor without changing the method: on Prism, the score of each of the 50 test cases; on Signal Processing, the nine raw metrics reported by the evaluator (composite score, slope changes, lag error, average error, false reversals, correlation, noise reduction, execution time, and success rate). On these two tasks, the two descriptors perform comparably (26.33 ± 0.10 vs. 26.50 ± 0.16 on Prism and 0.707 ± 0.075 vs. 0.713 ± 0.029 on Signal Processing; Table 5), indicating that the combined score is a sufficient descriptor for them. Whether this holds for other tasks, where programs with similar scores may differ more in behavior, remains to be tested.



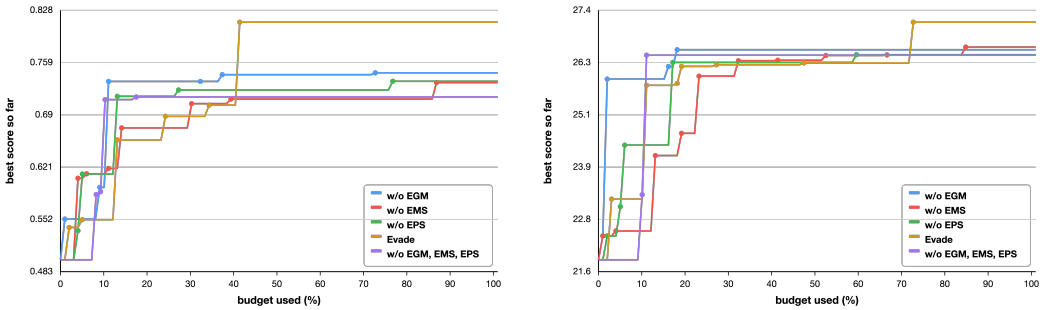
(a) GLM-5.2, Math

(b) GLM-5.2, ADRS

(c) DeepSeek-V4-Flash, Math

(d) DeepSeek-V4-Flash, ADRS

Figure 5: Behavioral redundancy rate (BRR, lower is better) on the mathematical and ADRS benchmarks with GLM-5.2 (top) and DeepSeek-V4-Flash (bottom). For GLM-5.2, each box aggregates the per-task mean BRR; for DeepSeek-V4-Flash, it aggregates per-run BRR over all tasks. EVADE consistently attains the lowest and most stable BRR.



(a) Signal Processing

(b) Prism

Figure 6: Optimization trajectories of EVADE and its ablated variants across the budget on (a) Signal Processing and (b) Prism. While ablated variants plateau early, EVADE avoids premature saturation and achieves decisive late-stage breakthroughs.

C.2 EXTENSION OF CASE STUDIES

C.2.1 REDUCING BEHAVIORAL REDUNDANCY ON SIGNAL PROCESSING AND PRISM

Figure 7 shows the search trees of AdaEvolve, ShinkaEvolve, and EVADE on Signal Processing and Prism. Both baselines spend much of their budget on redundant search, concentrating their mutations around a few high-fitness cells, yet they fail for different reasons. AdaEvolve repeatedly samples high-scoring programs and regenerates existing methods without exploring novel directions. ShinkaEvolve enforces syntactic code diversity, but still produces functionally equivalent behaviors. On Signal Processing, for example, AdaEvolve wastes search budget by repeatedly generating adap-

Table 5: Sensitivity to the hyperparameters of the evolvability signal and to the behavior descriptor, with DeepSeek-V4-Flash. Mean ± std over three runs. Each row changes one setting from the default.

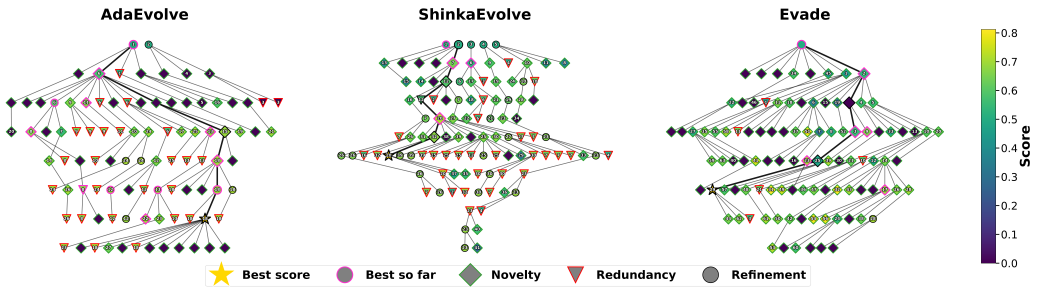
Setting	Prism (↑)	Signal Processing (↑)
Default ($\rho = 0.9$, $\kappa = 4$, $\alpha = 0.5$, combined score)	$\mathbf{26.50 \pm 0.16}$	$\mathbf{0.713 \pm 0.029}$
$\rho = 0.5$	26.31 ± 0.05	0.650 ± 0.029
$\rho = 0.6$	26.29 ± 0.06	0.679 ± 0.024
$\rho = 0.75$	26.42 ± 0.31	0.634 ± 0.019
$\rho = 0.95$	26.43 ± 0.31	0.703 ± 0.044
$\kappa = 1$	26.28 ± 0.04	0.647 ± 0.049
$\kappa = 2$	26.27 ± 0.18	0.635 ± 0.054
$\kappa = 8$	26.21 ± 0.08	0.656 ± 0.040
$\alpha = 0.25$	26.20 ± 0.10	0.649 ± 0.027
$\alpha = 0.75$	26.19 ± 0.06	0.666 ± 0.067
Vector descriptor	26.33 ± 0.10	0.707 ± 0.075

tive Holt filters and Savitzky–Golay variants; meanwhile, ShinkaEvolve creates one candidate that increases the Hodrick–Prescott (HP) filter penalty parameter while removing post-processing, and another that replaces a dense matrix solver with a sparse formulation—despite these implementation differences, both candidates yield identical output behaviors and fitness metrics. Consequently, BRR reaches 31.6% and 35.7% on Signal Processing and 82.0% and 40.4% on Prism for AdaEvolve and ShinkaEvolve, respectively, compared with only 5.7% and 10.0% for EVADE.

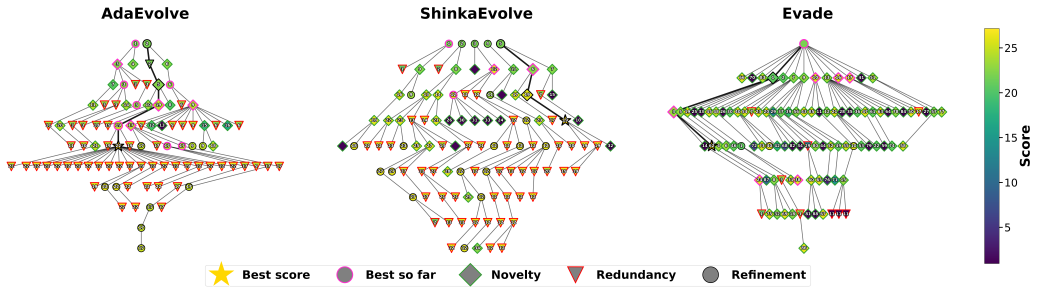
EVADE instead avoids wasting search budget by spreading its effort across several distinct lineages. On Signal Processing, the search moves from an adaptive Holt filter (0.542) through Savitzky–Golay variants (0.552, 0.657) to an L1 trend-filtering pipeline (0.688) and a spline trend-lock filter (0.703). At iteration 41, it selects a parent with a fitness of only 0.475, whose cell still has high evolvability. This parent mutates into a zero-phase Butterworth filter that reaches the best score of 0.812, a 15.5% gain over the previous best. On Prism, the search moves from a multi-start greedy placement (23.24) to simulated annealing (25.76), Hungarian assignment (25.80), and a MILP solver with bisection (26.18). At iteration 47, a best-fit candidate scoring only 20.74 advances the best score to 26.26, and at iteration 72, a candidate from iteration 3 mutates into a permutation search with hill climbing that reaches 27.16. In both runs, the best solution comes from a parent that a fitness-based rule would rarely select, because EVADE chooses parents by how much their cell can still improve rather than only by their current score or by code differences that do not directly measure the potential for novelty and improvement.

C.2.2 SURPASSING HUMAN BASELINES VIA ALGORITHMIC CONVERGENCE ON CLOUDCAST

On Cloudcast, which minimizes egress and instance costs for multi-cloud data broadcasts, EVADE progressively explores discrete routing paradigms to reach a cost of 618.09—beating both the Dijkstra baseline (1035) and prior human SOTA (626). The run advances through distinct algorithmic rungs rather than minor parameter tweaks: iterations 1–4 use ad-hoc path sharing ($1035 \rightarrow 954$), iteration 9 frames the task as a Steiner-tree problem via the Takahashi–Matsuyama heuristic (720), and iterations 16–20 add randomized GRASP restarts ($639 \rightarrow 627$). Earlier attempts to solve the problem exactly using generic LP flow relaxations failed (iterations 28 and 30 regressed up to 43.5%) because rounding fractional flows destroyed shared edges. When the search stalled at the 627–639 plateau, collapsing evolvability triggered breakthrough mode, generating the key insight at iteration 34: an exact Dreyfus–Wagner dynamic programming (DP) Steiner tree, made tractable by the small destination set (6–7 regions). Although this first attempt landed on a weak host program (897), the algorithm preserved the idea and reapplied it to a strong parent at iteration 39, reaching the global optimum of 618.09 (later matched bit-for-bit at iteration 48 by an exact MILP formulation). This cost marks the hard theoretical floor for the minimum-egress Steiner tree on this topology, showing how



(a) Signal Processing



(b) Prism

Figure 7: **Search trees of AdaEvolve, ShinkaEvolve, and EVADE** on (a) Signal Processing and (b) Prism. Each node is a behavioral cell, colored by fitness. Red triangles mark redundant mutations, whose child returns to a cell already on its lineage without improving on its parent; green diamonds mark mutations that reach a new cell. The star marks the best solution.

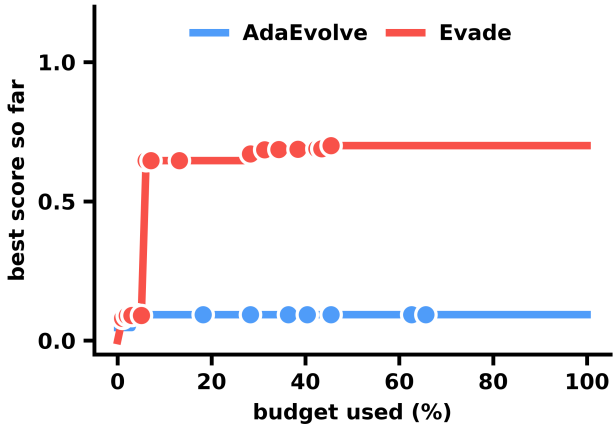


Figure 8: **Search trajectories on LLM-SQL** of AdaEvolve and EVADE. Both fall into an early local optimum ($\approx$ 0.08–0.09). AdaEvolve stays there for all 100 iterations, while EVADE escapes at iteration 6 and reaches 0.7001.

EVADE's evolvability-driven search discovers exact, human-competitive combinatorial algorithms once paired with a capable host lineage.

C.2.3 ESCAPING DEEP BASINS ON LLM-SQL

Figure 8 compares one run of AdaEvolve and one run of EVADE in which an early mutation leads both into a deep local optimum with a score of about 0.08–0.09. The trapped programs choose each

row's column order to match only the previous row, so the shared prefix breaks after a few rows and almost no prompt prefix is reused. Escaping requires a different program structure: a single column order for all rows, combined with multi-key row sorting. AdaEvolve never escapes and peaks at 0.0931. Its meta-guidance fires for 49 iterations, but it keeps selecting the highest-fitness program as the parent, so the LLM only produces small variations of the same sorting logic. EVADE escapes through selection rather than prompting. In breakthrough mode, it selects a low-fitness candidate (0.07) whose mutations keep reaching new cells, which raises its cell's evolvability from 0.60 to 0.84. Meanwhile, the local experience of this cell accumulates the strategies already tried, so later mutations avoid returning to the trap. This keeps the candidate selected long enough to discover the global column order at iteration 6, raising the score from 0.09 to 0.646 on all five datasets at once. The evolvability of the old basin then drops, parent selection moves to the new lineage, and the score reaches 0.7001 by iteration 54. Evolvability thus lets EVADE escape by allocating budget to the right candidate, while the accumulated experience keeps its mutations away from what has already failed.