WHAT DO EVOLUTIONARY CODING AGENTS EVOLVE?

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent work pairs LLMs with evolutionary search to iteratively generate, modify, and select code using task-specific feedback. These systems have produced strong results in mathematical discovery and algorithm design, yet a fundamental question remains: what do they actually evolve? Progress is typically summarized by the best score a run reaches under a task-specific evaluator, but that score can reflect several different mechanisms: new algorithmic structure, re-tuning an existing strategy, recombining ideas already in the model's internal knowledge, or overfitting to the evaluator. Distinguishing these mechanisms requires inspecting the search process itself, not only its final outcome. We introduce EvoTrace, a dataset of 121 evolutionary coding runs spanning four frameworks, five models, and 16 tasks across mathematics and algorithm design, comprising 10,672 unique programs. We also develop EvoReplay, which reconstructs local search states and supports held-out rescoring, same-prompt replay, and fixed-structure parameter tuning. Using a multi-label LLM judge validated against blind human re-annotation, we assign parent–child edits to nine recurring categories. The most frequent edit categories are not those most strongly associated with beating the parent. The median run reintroduces about 30% of its added lines from code deleted earlier in the same lineage, and these edits more often repair weak parents than advance the run best. Fixed-structure tuning can recover improvements otherwise reached through later LLM edits, exposed-evaluator gains can reverse on held-out instances, and same-prompt replay often recovers much of the original score without reproducing the winning code. These results show that benchmark gains in evolutionary coding agents can arise from qualitatively different mechanisms, only some of which correspond to new algorithmic structure.

1 INTRODUCTION

Large Language Model (LLM)-driven evolutionary code search uses LLMs as mutation operators within search loops over executable programs, guided by task-specific feedback (Romera-Paredes et al., 2024; Novikov et al., 2025; Georgiev et al., 2025; Lange et al., 2025; Assumpção et al., 2026; Cemri et al., 2026; Liu et al., 2026a). This paradigm has produced strong results across mathematical construction, systems optimization, algorithm design, and GPU kernel engineering, including improved bounds for combinatorial problems, better packing constructions, compiler and scheduling heuristics (Romera-Paredes et al., 2024; Novikov et al., 2025; Georgiev et al., 2025; Cheng et al., 2025a; Guo et al., 2025; Cao et al., 2026; Wiedemann et al., 2026). In this study, we define an **evolutionary coding agent** as a system with a task specification and executable evaluator, a population or archive of candidate programs, one or more LLMs that generate code mutations, recombinations, or refinements, and a search procedure that selects which programs and contexts feed future generations. A **search trace** is the full record produced by such a system: generated programs, scores, execution feedback, parent-child relations, prompts, model choices, and intermediate artifacts.

Despite rapid interest and empirical progress, the internal dynamics of evolutionary coding agents remain poorly understood. Existing work often reports final best scores, aggregate success rates, or a handful of illustrative trajectories, but such endpoints obscure the pathways and mechanisms by which improvements arise. While evolutionary coding agents sometimes demonstrate clear advantages over baselines such as independent sampling, greedy refinement, or beam-style search, these gains are highly sensitive to choices in task design, initialization, evaluator specification, or model configuration. There is consequently no clear consensus on what these systems are actually

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

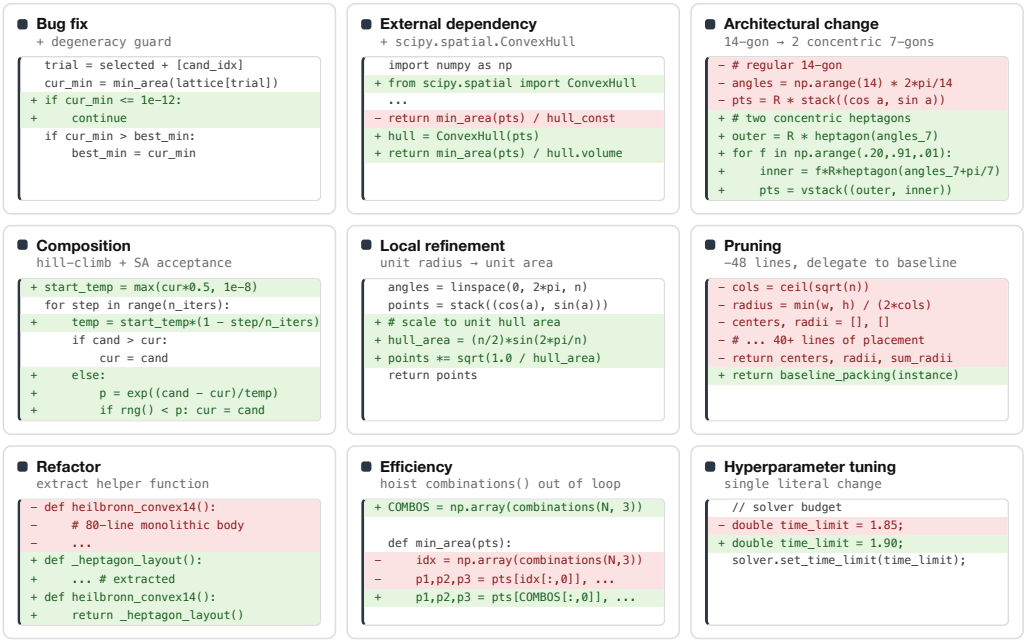


Figure 1: **A taxonomy of edits performed by evolutionary coding agents.** Each panel shows a representative parent–child diff (added lines in green, deleted lines in red) drawn from EvoTrace runs and labeled with one of nine recurring categories: *Bug fix*, *External dependency*, *Architectural change*, *Composition*, *Local refinement*, *Pruning*, *Refactor*, *Efficiency*, and *Hyperparameter tuning*. The categories range from minimal numeric edits (a single literal change) to structural rewrites (replacing a 14-gon with two concentric heptagons), and they form the basis of the LLM-as-judge edit annotation used throughout the paper. Edits are typically multi-label; we examine their prevalence and association with beating the parent in Section 5.5.

evolving: whether progress comes from discovering new program structure, tuning parameters of already-known strategies, recombining concepts already present in the model, or preserving early biases in the population. This motivates the central question we address: *What do evolutionary coding agents evolve, and how do their search dynamics produce improvements?*

To address this question, we introduce a dataset of evolutionary coding traces collected across multiple frameworks, models, and task families, covering mathematical constructions and algorithmic programming tasks. Rather than treating each run only by its final score, we study the full trajectory of generated programs: which solutions are explored, which lineages produce major improvements, and how stable those improvements are.

Analyzing these traces is non-trivial: a single run may generate hundreds of unique programs, and each candidate can differ from its ancestors through non-local structural changes, small numerical edits, prompt-driven rewrites, or evaluator-specific hacks. To make runs comparable, we develop a unified trace representation together with an annotation and measurement pipeline. The representation exposes the search graph, candidate programs, evaluations, and lineage information, enabling analyses of population structure, diversity, best-lineage stability, counterfactual model or context changes, and the decomposition of structural versus parametric gains.

We use this framework as a diagnostic tool for understanding both progress and failure in evolutionary coding. We first test whether apparent progress reflects repair of a weak parent (Section 5.1), survives held-out evaluation (Section 5.2), exceeds fixed-structure parameter tuning (Section 5.3), and can be reproduced from the same prompt (Section 5.4). We then characterize how programs, lineages, and edit types change during search (Section 5.5).

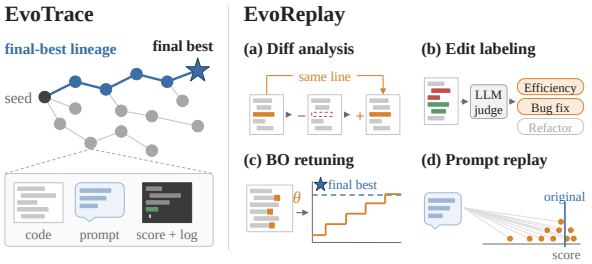


Figure 2: **EvoTrace and EvoReplay.** *Left:* EvoTrace stores each run as a search graph of evaluated programs (code, prompt, model, score, evaluator log); the blue path is the final-best lineage ($\star$). *Right:* EvoReplay (a) detects lines deleted and later re-added (Section 5.1), (b) labels edits with nine edit types (Section 5.5), (c) retunes an intermediate program's constants with Bayesian optimization (Section 5.3), and (d) resamples the saved prompts of best-so-far programs (Section 5.4).

These diagnostics are meant to be practical: the same trace representation and measurements can be integrated into existing open-source evolutionary coding frameworks to reveal whether a run is discovering new algorithmic structure, retuning known patterns, or becoming trapped by its own history. These measurements distinguish local code changes, replay stability, evaluator generalization, and fixed-structure tuning—properties that final-best scores conflate.

Contributions. We contribute two artifacts and a set of trace-level findings that use them. **(1) EvoTrace**, a dataset of 121 evolutionary coding-search runs across four frameworks on 16 benchmarks spanning Python mathematical constructions and C++ competitive programming problems, with 10,672 unique programs, 18,400 LLM calls, full parent–child graphs, prompts and contexts, scores, and evaluator metadata, normalized into a unified replayable schema (Section 3). **(2) EvoReplay**, a methodology and accompanying open-source package that reconstructs local search states from EvoTrace traces and reruns controlled interventions (same-prompt replay, Bayesian-optimization retuning, static analysis, deterministic cycling detection, ablation, repair, and context or model substitution), so that mechanistic claims about a search trajectory can be tested rather than asserted (Section 4). **(3) Findings** from applying these tools to characterize how evolutionary coding agents actually behave: reintroduced code often beats a weak parent without setting a new run best; public gains can reverse under held-out evaluation; Bayesian optimization of a fixed intermediate program can match later evolutionary gains; and re-running the same prompt rarely reproduces the winning code exactly (Section 5).

2 Related Work

2.1 LLM-Guided Evolutionary Coding Approaches

LLMs can act as mutation operators inside evolutionary loops over executable programs. FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025; Georgiev et al., 2025) established this paradigm, and a growing collection of open-source frameworks now extend it, including OpenEvolve (Sharma, 2025), GEPA (Agrawal et al., 2026), ShinkaEvolve (Lange et al., 2025), GigaEvo (Khrulkov et al., 2025), CodeEvolve (Assumpção et al., 2026), the FM Agent (Li et al., 2026a), and AIDE (Jiang et al., 2025b). A second wave targets the search procedure itself by adapting strategies, models, or signals during the run (Liu et al., 2026a; Cemri et al., 2026; Yan et al., 2026; Ray et al., 2026; Chen et al., 2026b), or by changing the unit of evolution to solution spaces, strategies, skills, prompt groups, populations, or concept trees (Zhai et al., 2025; Luo et al., 2026; Ye et al., 2026a; Li et al., 2025; Zhang et al., 2025; Leleu et al., 2026; Pourcel et al., 2026; Wang et al., 2025; Yuksekgonul et al., 2026; Ma et al., 2025; Singhal et al., 2026; Liu et al., 2026c).

A particularly active line of work targets GPU kernel optimization, where wall-clock runtime provides a tight reward signal (Guo et al., 2025; Wiedemann et al., 2026; Cao et al., 2026; Du et al., 2026; Nichols et al., 2026; Wei et al., 2025; Su et al., 2026; Cheng et al., 2025b;a). These frameworks have

also been applied to compiler heuristics, computer architecture, cosmology, swarm-intelligence design, symbolic regression, retrieval, recommendation, hyperparameter optimization, code optimization, agentic reasoning, autonomous data science, and broader scientific discovery (Chen et al., 2026a; Gupta et al., 2026; Li et al., 2026b; Cen & Tan, 2025; Song et al., 2025; Nian et al., 2026; Wang et al., 2026; Ferreira et al., 2026; Hu et al., 2026; Zhou et al., 2026; Yang et al., 2025; Weng et al., 2026; Liu et al., 2026b). This line builds on classical evolutionary computation and quality-diversity search (Koza, 1994; Jaderberg et al., 2017; Qian et al., 2024; Nadizar et al., 2025; Hughes et al., 2024); because the evolved objects are programs, recent work develops program-aware notions of diversity and similarity (Friedman & Dieng, 2023; Zhang & Lu, 2026). Our work is complementary: rather than proposing another framework, we analyze traces from four of them (OpenEvolve, GEPA, ShinkaEvolve, EvoX) to study the mechanisms by which programs improve, stagnate, diversify, or collapse.

2.2 Analyzing Evolutionary Coding Agents

A separate body of work studies evolutionary coding systems themselves. Surveys situate the broader space (Fang et al., 2025; Xie et al., 2025), benchmarks evaluate agents across ML competitions, long-horizon algorithm engineering, frontier research science, and production deployments (Chan et al., 2025; Imajuku et al., 2025; Lupidi et al., 2026; Ye et al., 2026b; Pan et al., 2026; Zheng et al., 2026), and Gideoni et al. (2026) show that simple baselines can match elaborate evolutionary pipelines. Closer to our methodology, several papers analyze search behavior rather than only final scores: trajectory analyses (Zhang et al., 2026), fitness-landscape characterization (Liu et al., 2025), failure modes of iterative LLM optimization (Nie et al., 2026), exploration deficits (Pan et al., 2025), output homogeneity (Jiang et al., 2025a), emergent risks in self-evolving agents (Shao et al., 2026), and a taxonomy of multi-agent failures (Cemri et al., 2025). Our work adopts a similar diagnostic perspective but focuses on full evolutionary coding traces, which we use to characterize how populations evolve, how improvement propagates through lineages, and how search dynamics produce both successes and failure modes.

3 EvoTrace

To understand what evolutionary coding agents evolve, we construct *EvoTrace*, a dataset of structured search traces from LLM-driven evolutionary coding systems. EvoTrace contains the artifacts needed to inspect and replay parts of the search process: generated programs, parent-child relations, prompts and retrieved context, evaluator outputs, scores, execution logs, and environment metadata. The dataset covers mathematical constructions and competitive programming tasks. These domains were chosen to capture distinct forms of code improvement: mathematical tasks reward new search algorithms, while competitive programming tasks require generated programs to compile and pass external judging, with limited access to the evaluator.

Different frameworks also log different parts of the search state, making direct cross-system comparison difficult. To address these issues, EvoTrace treats evolutionary search traces not merely as logs to annotate after the fact, but as structured computational objects that can be normalized, replayed, and intervened on. The collection and replay infrastructure is built on top of SkyDiscover (Liu et al., 2026b), a flexible framework for AI-driven scientific and algorithmic discovery; we extend it with a unified cross-backend schema, replay environments, and the analysis tooling described in Section 4.

3.1 Data Collection Across Tasks, Frameworks, and Models

We measure four evolutionary coding systems shown in Table 1. These are OpenEvolve (Sharma, 2025), GEPA (Agrawal et al., 2026), EvoX (Liu et al., 2026a), and ShinkaEvolve (Lange et al., 2025).

Table 1: Evolutionary coding frameworks in EvoTrace.

Framework	Search strategy	Distinctive feature
OpenEvolve	Archive-based, MAP-Elites style	Open-source reference baseline
GEPA	Reflective prompt evolution	Natural-language lessons drive mutation
EvoX	Meta-evolution	Adapts the search strategy during the run
ShinkaEvolve	Sample-efficient bandit search	Parent sampling, novelty rejection, model routing

EvoTrace covers 16 different tasks across two language–domain pairs, comprising 6 Python mathematical-discovery tasks (circle packing, Heilbronn placement, autocorrelation and uncertainty inequalities, signal processing) and 10 C++ competitive programming problems from ALE-bench Lite (Imajuku et al., 2025) (AtCoder Heuristic Contest), each with a judge-defined score. The four frameworks share the propose–evaluate–feedback pattern but differ in selection, context, diversity, and adaptation. We employ 5 different LLMs (Table 2) to generate the mutations and use 100 search iterations per run, yielding 121 runs and over 10,000 recorded program edits.

Table 2: Models used in EvoTrace.

Model	Role
`deepseek-reasoner`	Primary generator
`claude-sonnet-4-6`	Frontier alternative
`claude-haiku-4-5`	Small reasoning model
`gemini-3-flash-preview`	Cross-vendor alternative
`deepseek-chat`	Non-reasoning baseline

3.2 Trace schema and design choices

EvoTrace normalizes each run, regardless of which backend produced it, into a unified JSONL schema. The schema covers run-level metadata, candidate programs (full source, byte-identical), evaluator outputs (raw execution logs, errors, timings, task-specific metrics), parent–child edges with operator labels, the prompts and contexts the LLM saw at generation time, and the replay environment needed to rerun selected candidates against their original evaluator.

Recording the full source rather than a score-only log is what enables literal extraction for the BO baseline (Section 5.3), the cycling classifier (Section 5.1), and same-prompt replay (Section 5.4). Replayability is treated as a collection criterion: traces we cannot rerun against their original evaluator are excluded.

The schema supports two complementary uses: aggregate analysis (cross-framework comparison of population sizes, score progression, diversity, validity, lineage depth, best-so-far trajectories) and local reconstruction by EvoReplay (Section 4). The full per-field schema is given in Appendix A.1.

4 EvoReplay

We implement these analyses in *EvoReplay*, a Python package built on EvoTrace and SkyDiscover (Liu et al., 2026b). Each candidate program remains linked to its evaluator, parent context, and search position, allowing points in the search graph to be re-executed, perturbed, retuned, or re-judged and compared with the original run.

EvoReplay provides four capabilities on a common representation, making measurements and interventions comparable across frameworks and models.

(a) Static analysis of traces. EvoReplay normalizes runs from different backends into a common per-edit table containing the parent, child, prompt, score, and unified source diff. Program-size measurements, lineage depth, best-so-far trajectories, and the deterministic recycling classifier are then defined once and applied without framework-specific code paths.

(b) LLM-as-judge annotation of edit types. The 9-edit-type taxonomy was developed through manual inspection of parent–child edits sampled across frameworks, languages, and models. EvoRe-

play then applies it at scale: for every diff, an LLM judge returns one or more edit categories and the lines supporting those labels. The package handles batching, retries, schema validation, and caching, so a trace can be re-annotated with a different prompt or judge without rerunning the search.

(c) Bayesian optimization to study hyperparameter tuning. To separate structural change from numeric tuning, EvoReplay holds a program's structure fixed while retuning selected constants. One LLM call identifies candidate constants and ranges, after which Bayesian optimization evaluates 24 configurations with the program's original evaluator. The result measures how much improvement remains available from tuning alone. Implementation details and the full identification prompt are in Appendix B.9.

(d) Stability analysis of breakthroughs. EvoReplay can rerun the saved generating prompt for any candidate with the original or a substituted model and report the resulting distribution of children. It separately records whether each child parses, whether the evaluator accepts it, and its score when accepted. This distinguishes a robust search state from a single fortunate sample.

5 RESULTS

We analyze 121 evolutionary runs across the four frameworks of Table 1, 16 tasks spanning Python mathematical constructions and C++ ALE-bench problems (`ahc008`–`ahc046`), and 5 LLMs varied with and without diff-based generation (whether the LLM emits a unified diff or a full rewritten program). Each run consists of 100 search iterations. Aggregate program-, call-, and token-level statistics are reported in Appendix B.1. We treat beating a parent, setting a new run best, improving a held-out score, and reproducing a result as separate outcomes.

5.1 CYCLING AND RE-INTRODUCING PREVIOUSLY DELETED CODE

While manually inspecting traces to derive the edit taxonomy, we repeatedly observed lineages re-introducing lines they had earlier deleted, so we operationalized this as a deterministic check. For each parent–child diff, how often is an *added* line byte-identical to a line that the same lineage has already *deleted* in an earlier iteration? Across all 121 runs, the median per-run share of added lines that are such re-introductions is $\sim 30\%$, and this rate grows monotonically over the run in 118 of 121 cases (median per-iteration slope $+0.0030$). Recycling is present throughout nearly every measured trajectory and is dominated by short returns (median 5 iterations between deletion and re-introduction); the signal appears across all four frameworks, both languages, and all 5 generator models. A walk-through and additional breakdowns are in Appendices B.8 and B.6.

Reintroduced lines more often repair a weak parent than advance the run best. Structural edits recycle the most previously deleted code, with 37% of their added lines being reintroductions. Compared with other edits, edits for which reintroductions comprise most added lines are 17 percentage points more likely to beat their parent (95% CI $+11$ to $+24$), but 2.3 points less likely to establish a new run best (95% CI -4.1 to -0.5).

```
- SAFE_PENALTY = 1000
+ SAFE_PENALTY = 10000
- MAX_STUCK_TURNS = 10
+ MAX_STUCK_TURNS = 5
```

(i) Iteration 1

```
- SAFE_PENALTY = 10000
+ SAFE_PENALTY = 1000
- MAX_STUCK_TURNS = 5
+ MAX_STUCK_TURNS = 10
```

(ii) Iteration 4

Figure 3: **Recycling as repair.** In one EvoX `ahc008` lineage, (i) lowers the score from 3.67M to 2.43M; (ii) is a descendant that reintroduces 20 of its 22 added lines and returns to 3.67M, recovering the earlier best rather than setting a new one.

This line-level measure does not establish behavioral repetition. Exact whole-program returns account for only 5.33% of candidates. Occasional recycling can instead act as lineage-level error correction, allowing search to explore a damaging change and later recover useful code. Consistent with this interpretation, hard suppression did not improve final scores in ten additional paired runs. Recycling becomes diagnostic when combined with stagnation. A conservative retrospective plateau-plus-recycling rule saved 5.8% of evaluations, with no triggered run ending more than 0.5% below its eventual final score (Appendix B.6).

Finding. Occasional recycling acts as error correction; sustained recycling without a new run best indicates stagnation. Evolutionary agents should track new-best events separately from parent-relative gains and use plateau-plus-recycling as an early-stopping signal.

5.2 PUBLIC SCORES DO NOT ALWAYS GENERALIZE

ALE-bench public scores are not the held-out judging metric. Re-scoring every ALE run's public best-so-far chain on the private test set used by AtCoder (n=30 run/problem pairs across the four main backends), two of the four frameworks overfit on at least 30% of the problems they were scored on. The same problem can flip generalization sign across frameworks. On `ahc024`, OpenEvolve found a +1,606 rating-point private gain while ShinkaEvolve, on the same problem, *lost* 1,610 rating points despite a positive public score change. Full per-problem and per-framework tables are in Appendix B.7.

Finding. When agents optimize against an exposed evaluator, best-so-far scores may not transfer to held-out instances. Periodic held-out evaluation can reveal this gap.

5.3 THE TUNING GAP AND HYPERPARAMETER SEARCH

We separate a program p into a structure s and a hyperparameter vector $\theta \in \Theta_s$ it exposes, writing $p = s(\theta)$. Holding s_0 fixed, we run Bayesian optimization for 24 evaluator calls and denote its best observed score by $\hat{f}_{\mathrm{BO},24}(s_0)$. The *tuning gap* $\hat{\Delta}_{24}(s_0) = f^{\star}_{\mathrm{evo}} - \hat{f}_{\mathrm{BO},24}(s_0)$ compares the evolutionary run's best score with this finite-budget fixed-structure tuning result. The constant-selection prompt, parameter ranges, source rewrite, and optimizer configuration are in Appendix B.9.

BO matches the evolutionary run's final-best on most intermediate programs. We selected 36 programs by search position to cover early, intermediate, and final stages across runs, frameworks, and models; selection did not use the BO results. BO improves over the program's original score in 22 of 36 cases (Table 3). When compared against the run's *final*-best score, BO matches or exceeds it on 13 of 15 comparable intermediate programs (median delta +0.025). The median gain from tuning is +4.64% for early programs, +6.08% for selected intermediate programs, and +0.14% for final-best programs (Appendix B.9). The largest individual gain is on `heilbronn_tri_dsr_nodiff`, where the evolutionary run reached 0.521 in 100 iterations and BO on an intermediate program from the same run reached 0.886 (1.70× the evolutionary final best).

Table 3: BO outcomes on 36 programs sampled across search stages (median 6 knobs per program; 24 evaluator calls each).

Outcome	# targets
BO improves over original	22
No change	8
BO regresses	6
Total	**36**

Finding. Fixed-structure tuning can recover gains otherwise obtained through later evolutionary edits. Evolutionary coding agents should use LLM edits to find useful structures, then hand exposed constants to a dedicated optimizer.

5.4 REPLAY REPRODUCIBILITY WITHOUT LEXICAL REPRODUCTION

When evolutionary search reaches a new best-so-far program, can we reproduce that breakthrough by re-running the same prompt? For each of 36 best-so-far events across the four backends, we re-prompted an LLM 10 times with the exact context the original run had used and asked four questions of each replayed program. Does it run? Does the evaluator accept it? Does it match the original program byte-for-byte? Does it match the original score? Table 4 reports the medians and four illustrative targets covering the recurring patterns.

Same-prompt replay distinguishes a robust search state from a single fortunate sample. Important breakthroughs should be reported with a distribution of replayed scores, rather than only one winning program and score.

Table 4: **Replay summary across** 36 **breakthrough events.** Aggregate medians and four illustrative targets covering recurring replay patterns. "Replay/Original" is the median replayed score divided by the original program's score.

Target	Parse	Eval	Exact	Replay/Orig.	Pattern
Median across 36 **targets**	1.00	1.00	0.00	0.76	—
`circle_packing_iter68_improve`	1.00	1.00	≥ 0	≈ 1.00	tight reproduction
`second_autocorr_iter79_improve`	1.00	1.00	0.00	0.96	parent-revert
`circle_packing_iter32_neutral`	0.50	0.50	0.00	—	bimodal failure
`heilbronn_convex_iter100_regress`	1.00	1.00	0.00	1.11	replays exceed regress.

Finding. Replays almost always produce a runnable program (median parse and evaluator success 1.00) but essentially never the original program (median exact-match 0.00). They nevertheless recover a median 0.76 of the original score from a *different* program. The score gain is broadly reproducible from the same prompt context even though the specific program is not.

5.5 What changes during search?

Programs change shape during search (Figure 4). Math runs add code and expose more numeric constants, while ALE runs refine already-large seeds at near-constant size. The plotted quantity is the count of numeric literals relative to the seed, not growth in their values.

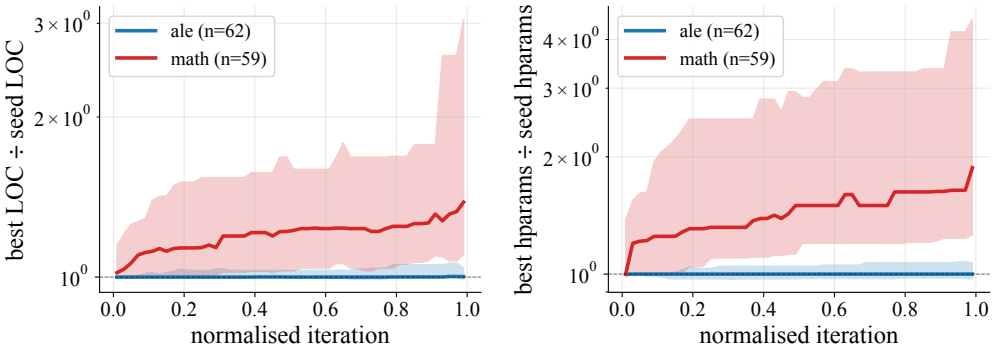
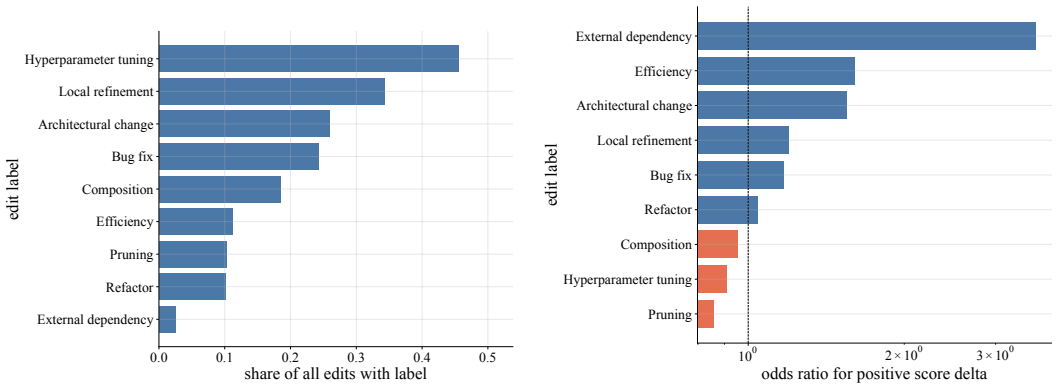


Figure 4: **Program size and numeric-literal hyperparameter count over a run.** Best-so-far program length (LOC, left) and numeric-literal count (right), each normalized by the run's seed value, plotted against normalized iteration. Solid line = cross-run median; shaded band = inter-quartile range; dashed gray line marks the seed value. Math runs (n=59) accumulate code and tunable constants (median final ratios 1.33× and 1.70×); ALE runs (n=62) refine large seeds in place (final ratio $\approx$ 1.0× on both axes).

Lineage depth and budget utilization. The chain of parents from each final-best program back to the seed is short in both domains, but somewhat shorter on ALE (median lineage depth 4, vs. 6 on math). The final-best program first appears at a median normalized iteration of 0.75 on math and 0.49 on ALE. Most evaluated programs are not on the final winner's direct parent chain. We call them *off-lineage evaluations*; they may still affect selection, archive diversity, or information supplied in later prompts.

Edit taxonomy via LLM-as-judge. We use EvoReplay's LLM-as-judge pipeline (Section 4, capability (b)) to annotate every parent–child edit with one or more categories from a 9-label taxonomy comprising *Hyperparameter tuning*, *Local refinement*, *Architectural change*, *Composition*, *Efficiency*, *Bug fix*, *External dependency*, *Pruning*, and *Refactor*. Agreement between the judge and a blind human re-annotation on a stratified sample of 200 edits is substantial overall (macro $\kappa = 0.77$,



(a) Prevalence

(b) Beats parent (odds ratio)

Figure 5: **Edit frequency and parent-relative improvement** across EvoTrace. **(a)** Frequency of each label. **(b)** Odds ratio that an edit beats its parent. Best-so-far and final-best-lineage enrichment views, plus per-domain and per-backend breakdowns, are in Appendices B.2 and B.3.

micro-$F_1 = 0.90$, exact-match accuracy 74.5%). Working definitions, examples, and the ambiguity caused by multi-label edits are reported in Appendix B.4.

Hyperparameter tuning is the most prevalent label. The strongest associations with beating the parent occur for *External dependency* ($3.58\times$ odds ratio; n=104), *Efficiency* ($1.61\times$; n=464), and *Architectural change* ($1.55\times$; n=1,075). Here, improvement means beating the parent, not necessarily setting a new run best. Edits are typically multi-label (67.4% have ≥ 2 labels), so the categories are not mutually exclusive.

The distribution of edits differs from the distribution of locally successful edits. Tuning and local refinement are most common, whereas external dependencies, efficiency changes, and architectural changes have the highest odds of beating their parent. These categories are opportunity-dependent. One structural change may enable many subsequent refinements, so the odds ratios do not define a preferred edit frequency.

6 Discussion

The results support four evaluation practices. Recycling should be monitored rather than blocked, and its coincidence with a score plateau can provide a conservative stopping signal. Improvements on an exposed evaluator should be checked periodically on held-out instances. For programs with exposed constants, structural search should be compared with a fixed-budget tuning baseline. Important breakthroughs should be accompanied by a replayed score distribution rather than a single winning sample. These diagnostics distinguish repair, tuning, evaluator adaptation, and reproducible progress without requiring another full search.

Limitations. The numerical rates describe the frameworks, models, and tasks in EvoTrace and may change for newer systems. Edit labels are observational and multi-label, so their associations with score changes are not causal. Fixed-structure tuning is evaluated only on math, private rescoring only on ALE, and replay uses ten samples per event. EvoReplay allows these tests to be repeated as new traces become available.

AI Use Statement

Generative AI tools were used to assist with code development and language editing. All AI-assisted outputs were reviewed by the authors, who take responsibility for the paper's content.

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, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning, February 2026. URL https://arxiv.org/abs/2507.19457.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. CodeEvolve: An open source evolutionary coding agent for algorithmic discovery and optimization, March 2026. URL https://arxiv.org/abs/2510.14150.

Shiyi Cao, Ziming Mao, Joseph E. Gonzalez, and Ion Stoica. K-Search: LLM Kernel Generation via Co-Evolving Intrinsic World Model, February 2026. URL https://arxiv.org/abs/2602.19128.

Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why Do Multi-Agent LLM Systems Fail?, October 2025. URL https://arxiv.org/abs/2503.13657.

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

Shipeng Cen and Ying Tan. Beyond Algorithm Evolution: An LLM-Driven Framework for the Co-Evolution of Swarm Intelligence Optimization Algorithms and Prompts, December 2025. URL https://arxiv.org/abs/2512.09209.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering, February 2025. URL https://arxiv.org/abs/2410.07095.

Hongzheng Chen, Alexander Novikov, Ngân Vũ, Hanna Alam, Zhiru Zhang, Aiden Grossman, Mircea Trofin, and Amir Yazdanbakhsh. Magellan: Autonomous Discovery of Novel Compiler Optimization Heuristics with AlphaEvolve, January 2026a. URL https://arxiv.org/abs/2601.21096.

Yongqiang Chen, Chenxi Liu, Zhenhao Chen, Tongliang Liu, Bo Han, and Kun Zhang. CausalEvolve: Towards Open-Ended Discovery with Causal Scratchpad, March 2026b. URL https://arxiv.org/abs/2603.14575.

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, December 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, October 2025b. URL https://arxiv.org/abs/2510.06189.

Weihua Du, Jingming Zhuo, Yixin Dong, Andre Wang He, Weiwei Sun, Zeyu Zheng, Manupa Karunaratne, Ivan Fox, Tim Dettmers, Tianqi Chen, Yiming Yang, and Sean Welleck. AdaExplore: Failure-Driven Adaptation and Diversity-Preserving Search for Efficient Kernel Generation, April 2026. URL https://arxiv.org/abs/2604.16625.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. A Comprehensive Survey of Self-Evolving AI Agents: A New Paradigm Bridging Foundation

Models and Lifelong Agentic Systems, August 2025. URL https://arxiv.org/abs/2508.07407.

Fabio Ferreira, Lucca Wobbe, Arjun Krishnakumar, Frank Hutter, and Arber Zela. Can LLMs Beat Classical Hyperparameter Optimization Algorithms? A Study on autoresearch, April 2026. URL https://arxiv.org/abs/2603.24647.

Dan Friedman and Adji Bousso Dieng. The Vendi Score: A Diversity Evaluation Metric for Machine Learning, July 2023. URL https://arxiv.org/abs/2210.02410.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale, December 2025. URL https://arxiv.org/abs/2511.02864.

Yonatan Gideoni, Sebastian Risi, and Yarin Gal. Simple Baselines are Competitive with Code Evolution, February 2026. URL https://arxiv.org/abs/2602.16805.

Ping Guo, Chenyu Zhu, Siyuan Chen, Fei Liu, Xi Lin, Zhichao Lu, and Qingfu Zhang. EvoEngineer: Mastering Automated CUDA Kernel Code Evolution with Large Language Models, October 2025. URL https://arxiv.org/abs/2510.03760.

Raghav Gupta, Akanksha Jain, Abraham Gonzalez, Alexander Novikov, Po-Sen Huang, Matej Balog, Marvin Eisenberger, Sergey Shirobokov, Ngân Vũ, Martin Dixon, Borivoje Nikolić, Parthasarathy Ranganathan, and Sagar Karandikar. ArchAgent: Agentic AI-driven Computer Architecture Discovery, February 2026. URL https://arxiv.org/abs/2602.22425.

Tu Hu, Ronghao Chen, Shuo Zhang, Jianghao Yin, Mou Xiao Feng, Jingping Liu, Shaolei Zhang, Wenqi Jiang, Yuqi Fang, Sen Hu, Huacan Wang, and Yi Xu. Controlled Self-Evolution for Algorithmic Code Optimization, February 2026. URL https://arxiv.org/abs/2601.07348.

Edward Hughes, Michael Dennis, Jack Parker-Holder, Feryal Behbahani, Aditi Mavalankar, Yuge Shi, Tom Schaul, and Tim Rocktaschel. Open-Endedness is Essential for Artificial Superhuman Intelligence, June 2024. URL https://arxiv.org/abs/2406.04268.

Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. ALE-Bench: A Benchmark for Long-Horizon Objective-Driven Algorithm Engineering, October 2025. URL https://arxiv.org/abs/2506.09050.

Max Jaderberg, Valentin Dalibard, Simon Osindero, Wojciech M. Czarnecki, Jeff Donahue, Ali Razavi, Oriol Vinyals, Tim Green, Iain Dunning, Karen Simonyan, Chrisantha Fernando, and Koray Kavukcuoglu. Population Based Training of Neural Networks, November 2017. URL https://arxiv.org/abs/1711.09846.

Liwei Jiang, Yuanjun Chai, Margaret Li, Mickel Liu, Raymond Fok, Nouha Dziri, Yulia Tsvetkov, Maarten Sap, Alon Albalak, and Yejin Choi. Artificial Hivemind: The Open-Ended Homogeneity of Language Models (and Beyond), October 2025a. URL https://arxiv.org/abs/2510.22954.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. AIDE: AI-Driven Exploration in the Space of Code, February 2025b. URL https://arxiv.org/abs/2502.13138.

Valentin Khrulkov, Andrey Galichin, Denis Bashkirov, Dmitry Vinichenko, Oleg Travkin, Roman Alferov, Andrey Kuznetsov, and Ivan Oseledets. GigaEvo: An Open Source Optimization Framework Powered By LLMs And Evolution Algorithms, November 2025. URL https://arxiv.org/abs/2511.17592.

JohnR. Koza. Genetic programming as a means for programming computers by natural selection. *Statistics and Computing*, 4(2), June 1994. ISSN 0960-3174, 1573-1375. doi: 10.1007/BF00175355.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution, September 2025. URL https://arxiv.org/abs/2509.19349.

Timothee Leleu, Sudeera Gunathilaka, Federico Ghimenti, and Surya Ganguli. Contrastive Concept-Tree Search for LLM-Assisted Algorithm Discovery, February 2026. URL https://arxiv.org/abs/2602.03132.

Annan Li, Chufan Wu, Zengle Ge, Yee Hin Chong, Zhinan Hou, Lizhe Cao, Cheng Ju, Jianmin Wu, Huaiming Li, Haobo Zhang, Shenghao Feng, Mo Zhao, Fengzhi Qiu, Rui Yang, Mengmeng Zhang, Wenyi Zhu, Yingying Sun, Quan Sun, Shunhao Yan, Danyu Liu, Dawei Yin, and Dou Shen. The FM Agent, February 2026a. URL https://arxiv.org/abs/2510.26144.

Tiancheng Li, Yuhang Wang, Zhiyang Chen, Zijun Wang, Liyuan Ma, and Guo-jun Qi. C-Evolve: Consensus-based Evolution for Prompt Groups, September 2025. URL https://arxiv.org/abs/2509.23331.

Tianyi Li, Shihui Zang, and Moritz Münchmeyer. MadEvolve: Evolutionary Optimization of Cosmological Algorithms with Large Language Models, February 2026b. URL https://arxiv.org/abs/2602.15951.

Fei Liu, Qingfu Zhang, Jialong Shi, Xialiang Tong, Kun Mao, and Mingxuan Yuan. Fitness Landscape of Large Language Model-Assisted Automated Algorithm Search, August 2025. URL https://arxiv.org/abs/2504.19636.

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, March 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, New York, NY, USA, 2026b. Association for Computing Machinery. ISBN 9798400724152. doi: 10.1145/3786335.3813221. URL https://doi.org/10.1145/3786335.3813221.

Wei Liu, Siya Qi, Yali Du, and Yulan He. Self-play only evolves when self-synthetic pipeline ensures learnable information gain, 2026c. URL https://arxiv.org/abs/2603.02218.

Sichun Luo, Yi Huang, Haochen Luo, Fengyuan Liu, Guanzhi Deng, Lei Li, Qinghua Yao, Zefa Hu, Junlan Feng, and Qi Liu. SeaEvo: Advancing Algorithm Discovery with Strategy Space Evolution, April 2026. URL https://arxiv.org/abs/2604.24372.

Alisia Lupidi, Bhavul Gauri, Thomas Simon Foster, Bassel Al Omari, Despoina Magka, Alberto Pepe, Alexis Audran-Reiss, Muna Aghamelu, Nicolas Baldwin, Lucia Cipolina-Kun, Jean-Christophe Gagnon-Audet, Chee Hau Leow, Sandra Lefdal, Hossam Mossalam, Abhinav Moudgil, Saba Nazir, Emanuel Tewolde, Isabel Urrego, Jordi Armengol Estape, Amar Budhiraja, Gaurav Chaurasia, Abhishek Charnalia, Derek Dunfield, Karen Hambardzumyan, Daniel Izcovich, Martin Josifoski, Ishita Mediratta, Kelvin Niu, Parth Pathak, Michael Shvartsman, Edan Toledo, Anton Protopopov, Roberta Raileanu, Alexander Miller, Tatiana Shavrina, Jakob Foerster, and Yoram Bachrach. AIRS-Bench: A Suite of Tasks for Frontier AI Research Science Agents, February 2026. URL https://arxiv.org/abs/2602.06855.

Ruiying Ma, Chieh-Jan Mike Liang, Yanjie Gao, and Francis Y. Yan. MetaMuse: Algorithm Generation via Creative Ideation, October 2025. URL https://arxiv.org/abs/2510.03851.

Giorgia Nadizar, Francesco Rusin, Eric Medvet, and Gabriela Ochoa. The Role of Stepping Stones in MAP-Elites: Insights from Search Trajectory Networks. In Bing Xue, Luca Manzoni, and Illya Bakurov (eds.), *Genetic Programming*, volume 15609, pp. 224–239. Springer Nature Switzerland, Cham, 2025. ISBN 978-3-031-89990-4 978-3-031-89991-1. doi: 10.1007/978-3-031-89991-1_14.

Jinming Nian, Fangchen Li, Dae Hoon Park, and Yi Fang. RankEvolve: Automating the Discovery of Retrieval Algorithms via LLM-Driven Evolution, February 2026. URL https://arxiv.org/abs/2602.16932.

Daniel Nichols, Konstantinos Parasyris, Caetano Melone, Tal Ben-Nun, Giorgis Georgakoudis, and Harshitha Menon. Record-Remix-Replay: Hierarchical GPU Kernel Optimization using Evolutionary Search, April 2026. URL https://arxiv.org/abs/2604.11109.

Allen Nie, Xavier Daull, Zhiyi Kuang, Abhinav Akkiraju, Anish Chaudhuri, Max Piasevoli, Ryan Rong, YuCheng Yuan, Prerit Choudhary, Shannon Xiao, Rasool Fakoor, Adith Swaminathan, and Ching-An Cheng. Understanding the Challenges in Iterative Generative Optimization with LLMs, March 2026. URL https://arxiv.org/abs/2603.23994.

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, June 2025. URL https://arxiv.org/abs/2506.13131.

Lan Pan, Hanbo Xie, and Robert C. Wilson. Large Language Models Think Too Fast To Explore Effectively, May 2025. URL https://arxiv.org/abs/2501.18009.

Melissa Z. Pan, Negar Arabzadeh, Riccardo Cogo, Yuxuan Zhu, Alexander Xiong, Lakshya A. Agrawal, Huanzhi Mao, Emma Shen, Sid Pallerla, Liana Patel, Shu Liu, Tianneng Shi, Xiaoyuan Liu, Jared Quincy Davis, Emmanuele Lacavalla, Alessandro Basile, Shuyi Yang, Paul Castro, Daniel Kang, Joseph E. Gonzalez, Koushik Sen, Dawn Song, Ion Stoica, Matei Zaharia, and Marquita Ellis. Measuring Agents in Production, February 2026. URL https://arxiv.org/abs/2512.04123.

Julien Pourcel, Cédric Colas, and Pierre-Yves Oudeyer. Self-Improving Language Models for Evolutionary Program Synthesis: A Case Study on ARC-AGI, March 2026. URL https://arxiv.org/abs/2507.14172.

Chao Qian, Ke Xue, and Ren-Jian Wang. Quality-Diversity Algorithms Can Provably Be Helpful for Optimization, May 2024. URL https://arxiv.org/abs/2401.10539.

Pretam Ray, Pratik Prabhanjan Brahma, Zicheng Liu, and Emad Barsoum. AdaptEvolve: Improving Efficiency of Evolutionary AI Agents through Adaptive Model Selection, February 2026. URL https://arxiv.org/abs/2602.11931.

Bernardino Romera-Paredes, Mohammadamin Barekatain, 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, January 2024. ISSN 0028-0836, 1476-4687. doi: 10.1038/s41586-023-06924-6.

Shuai Shao, Qihan Ren, Chen Qian, Boyi Wei, Dadi Guo, Jingyi Yang, Xinhao Song, Linfeng Zhang, Weinan Zhang, Dongrui Liu, and Jing Shao. Your Agent May Misevolve: Emergent Risks in Self-evolving LLM Agents, March 2026. URL https://arxiv.org/abs/2509.26354.

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

Shivam Singhal, Priyadarsi Mishra, Eran Malach, and Tomer Galanti. LLM Priors for ERM over Programs, February 2026. URL https://arxiv.org/abs/2510.14331.

Zhuo-Yang Song, Zeyu Cai, Shutao Zhang, Jiashen Wei, Jichen Pan, Shi Qiu, Qing-Hong Cao, Tie-Jiun Hou, Xiaohui Liu, Ming-xing Luo, and Hua Xing Zhu. Iterated Agent for Symbolic Regression, October 2025. URL https://arxiv.org/abs/2510.08317.

Hongyuan Su, Yu Zheng, and Yong Li. ContextEvolve: Multi-Agent Context Compression for Systems Code Optimization, February 2026. URL https://arxiv.org/abs/2602.02597.

Haochen Wang, Yi Wu, Daryl Chang, Li Wei, and Lukasz Heldt. Self-Evolving Recommendation System: End-To-End Autonomous Model Optimization With LLM Agents, February 2026. URL https://arxiv.org/abs/2602.10226.

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, November 2025. URL https://arxiv.org/abs/2511.23473.

Anjiang Wei, Tianran Sun, Yogesh Seenichamy, Hang Song, Anne Ouyang, Azalia Mirhoseini, Ke Wang, and Alex Aiken. Astra: A Multi-Agent System for GPU Kernel Performance Optimization, December 2025. URL https://arxiv.org/abs/2509.07506.

Zhaotian Weng, Antonis Antoniades, Deepak Nathani, Zhen Zhang, Xiao Pu, and Xin Eric Wang. Group-Evolving Agents: Open-Ended Self-Improvement via Experience Sharing, February 2026. URL https://arxiv.org/abs/2602.04837.

Nina Wiedemann, Quentin Leboutet, Michael Paulitsch, Diana Wofk, and Benjamin Ummenhofer. KernelFoundry: Hardware-aware evolutionary GPU kernel optimization, March 2026. URL https://arxiv.org/abs/2603.12440.

Qiujie Xie, Yixuan Weng, Minjun Zhu, Fuchen Shen, Shulin Huang, Zhen Lin, Jiahui Zhou, Zilan Mao, Zijie Yang, Linyi Yang, Jian Wu, and Yue Zhang. How Far Are AI Scientists from Changing the World?, August 2025. URL https://arxiv.org/abs/2507.23276.

Minghao Yan, Bo Peng, Benjamin Coleman, Ziqi Chen, Zhouhang Xie, Shuo Chen, Zhankui He, Noveen Sachdeva, Isabella Ye, Weili Wang, Chi Wang, Ed H. Chi, Fernando Pereira, Wang-Cheng Kang, Derek Zhiyuan Cheng, and Beidou Wang. PACEvolve: Enabling Long-Horizon Progress-Aware Consistent Evolution, January 2026. URL https://arxiv.org/abs/2601.10657.

Xu Yang, Xiao Yang, Shikai Fang, Yifei Zhang, Jian Wang, Bowen Xian, Qizheng Li, Jingyuan Li, Minrui Xu, Yuante Li, Haoran Pan, Yuge Zhang, Weiqing Liu, Yelong Shen, Weizhu Chen, and Jiang Bian. R&D-Agent: An LLM-Agent Framework Towards Autonomous Data Science, October 2025. URL https://arxiv.org/abs/2505.14738.

Haoran Ye, Xuning He, Vincent Arak, Haonan Dong, and Guojie Song. Meta Context Engineering via Agentic Skill Evolution, February 2026a. URL https://arxiv.org/abs/2601.21557.

Haotian Ye, Haowei Lin, Jingyi Tang, Yizhen Luo, Caiyin Yang, Chang Su, Rahul Thapa, Rui Yang, Ruihua Liu, Zeyu Li, Chong Gao, Dachao Ding, Guangrong He, Miaolei Zhang, Lina Sun, Wenyang Wang, Yuchen Zhong, Zhuohao Shen, Di He, Jianzhu Ma, Stefano Ermon, Tongyang Li, Xiaowen Chu, James Zou, and Yuzhi Xu. Evaluation-driven Scaling for Scientific Discovery, April 2026b. URL https://arxiv.org/abs/2604.19341.

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, February 2026. URL https://arxiv.org/abs/2601.16175.

Yi Zhai, Zhiqiang Wei, Ruohan Li, Keyu Pan, Shuo Liu, Lu Zhang, Jianmin Ji, Wuyang Zhang, Yu Zhang, and Yanyong Zhang. \(X\)-evolve: Solution space evolution powered by large language models, August 2025. URL https://arxiv.org/abs/2508.07932.

Rui Zhang and Zhichao Lu. Rethinking Code Similarity for Automated Algorithm Design with LLMs, March 2026. URL https://arxiv.org/abs/2603.02787.

Xinhao Zhang, Xi Chen, François Portet, and Maxime Peyrard. What Makes an LLM a Good Optimizer? A Trajectory Analysis of LLM-Guided Evolutionary Search, April 2026. URL https://arxiv.org/abs/2604.19440.

Yanzhi Zhang, Yitong Duan, Zhaoxi Zhang, Jiyan He, and Shuxin Zheng. Population-Evolve: A Parallel Sampling and Evolutionary Method for LLM Math Reasoning, December 2025. URL https://arxiv.org/abs/2512.19081.

Jingsheng Zheng, Jintian Zhang, Yujie Luo, Yuren Mao, Yunjun Gao, Lun Du, Huajun Chen, and Ningyu Zhang. Can We Predict Before Executing Machine Learning Agents?, January 2026. URL https://arxiv.org/abs/2601.05930.

Zhanke Zhou, Chentao Cao, Xiao Feng, Xuan Li, Zongze Li, Xiangyu Lu, Jiangchao Yao, Weikai Huang, Tian Cheng, Jianghangfan Zhang, Tangyu Jiang, Linrui Xu, Yiming Zheng, Brando Miranda, Tongliang Liu, Sanmi Koyejo, Masashi Sugiyama, and Bo Han. AlphaApollo: A System for Deep Agentic Reasoning, March 2026. URL https://arxiv.org/abs/2510.06261.

A Additional EvoTrace Details

A.1 Per-field trace schema

EvoTrace normalizes each run into six object types stored as JSONL tables, each motivated by a mechanistic question that storing only iteration-vs-score traces would foreclose. Table 5 summarises the six object types.

Table 5: EvoTrace per-field schema. Each object type is recorded because at least one analysis in the main paper requires the corresponding raw artifact rather than a score-only summary.

Object	**Fields**	**Why recorded**
Runs	Task, framework, model configuration, evaluator command, seed artifact, search budget, run-time environment.	Makes any reported number sliceable by framework or model.
Candidates	Full program source (byte-identical), iteration index, parent identifiers, prompt context, validity status, evaluator score.	The full source enables literal extraction for the BO baseline (Section 5.3, Appendix B.9), the cycling classifier (Section 5.1), and post-hoc simplification or repair.
Evaluations	Execution outputs, error messages, timing, correctness signals, task-specific metrics.	The bimodal failure picture in Section 5.4 is invisible to score-only logging.
Edges	Parent–child relations with the operator that produced the child (mutation, recombination, refinement, repair).	Lineage-depth and dead-branch statistics in Section 5.5 require the full edge table, not the best-so-far trajectory.
Contexts	Prompts, retrieved examples, population summaries, and lineage information seen by the LLM at generation time (byte-identical).	Stability replays in Section 5.4 pass the saved input back to the original or a substituted model.
Replay environments	Evaluator command, dependencies, timeouts, hardware assumptions, raw artifacts.	Replayability is a collection criterion: traces we cannot rerun against the original evaluator are excluded.

B Additional Experimental Details

B.1 Experiment scale and cost

The dataset spans 121 evolutionary runs that together propose 10,672 unique programs (including 1,708 explicitly rejected ones), make 18,400 LLM calls, and consume 274.7 M prompt and 80.3 M completion tokens (of which 42.8 M are reasoning tokens). A typical 100-iteration run produces about 100 programs, makes about 134 LLM calls, and uses $\sim$ 1.7 M prompt and $\sim$ 535 K completion tokens. ALE runs use approximately 2.8$\times$ more prompt tokens than math runs because of their much larger seeds. Tables 6, 7, and 8 report full breakdowns.

Table 6: Experiment scale by backend. "Edits" counts programs with a non-null parent; LLM calls and tokens are aggregated from each run's call log.

backend	runs	programs	accepted	rejected	edits	LLM calls	prompt tok	compl. tok
openevolve	44	4,267	4,267	0	4,223	5,064	84.9 M	21.1 M
evox	30	2,396	2,396	0	2,366	4,700	71.5 M	14.4 M
gepa	29	2,137	429	1,708	2,108	4,194	78.1 M	19.7 M
shinka	18	1,872	1,872	0	1,782	4,442	40.2 M	25.1 M
total	**121**	**10,672**	**8,964**	**1,708**	**10,479**	**18,400**	**274.7 M**	**80.3 M**

Table 7: Experiment scale by domain (ALE vs. math).

domain	runs	programs	edits	LLM calls	prompt tok	compl. tok
ale	56	5,269	5,173	9,341	201.7 M	42.8 M
math	65	5,403	5,306	9,059	73.0 M	37.5 M
total	**121**	**10,672**	**10,479**	**18,400**	**274.7 M**	**80.3 M**

Table 8: Per-run cost (medians and right tails) over the 121 runs.

per-run metric	median	75th pct	max
programs	101	107	118
edits (parent → child)	99	106	117
LLM calls	134	165	512
prompt tokens	1.7 M	3.3 M	7.3 M
completion tokens	535,582	811,225	2.7 M

B.2 Edit-taxonomy: aggregate enrichment views

The main paper (Figure 5) compares edit prevalence with the odds that an edit beats its parent. The companion enrichment panels are reported here. Relative to the all-edits base rate, best-so-far updates are enriched in *Efficiency* (1.49×), *External dependency* (1.34×), and *Hyperparameter tuning* (1.32×); final-best lineages retain a similar mix, with *Efficiency* (1.42×), *Hyperparameter tuning* (1.27×), and *Composition* (1.21×) overrepresented (Figures 6 and 7).

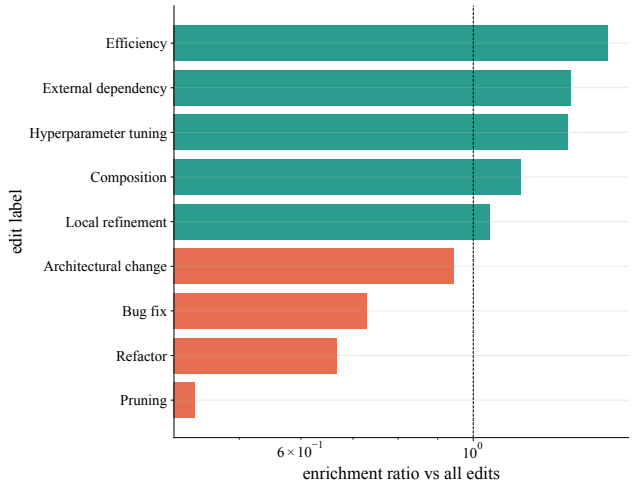


Figure 6: **Best-so-far enrichment of edit labels (aggregate).** Enrichment of each taxonomy label among best-so-far updates relative to the all-edits base rate. The categories most overrepresented on successful intermediate steps (*Efficiency*, *External dependency*, *Hyperparameter tuning*, *Composition*) are not identical to the most frequent labels in Figure 5 (a).

B.3 Edit-taxonomy breakdowns by domain and backend

The aggregate edit-taxonomy results reported in Section 5.5 (Figure 5) combine ALE and math runs across all four backends. This section reports the same analyses split by domain and, for the parent-relative view, by backend. The underlying labeled corpus covers all programs in EvoTrace.

Table 9: **Strength of EvoTrace construction runs relative to published AlphaEvolve results.** Scores are normalized by the corresponding AlphaEvolve reference, so 1.0 matches the published result. The three signal-processing runs use a different composite score and are excluded.

Statistic	Value
Runs with a corresponding AlphaEvolve reference	62
Median normalized best score	0.9806
Runs ≥ 0.90	43
Runs ≥ 0.98	32
Runs ≥ 1.00	14

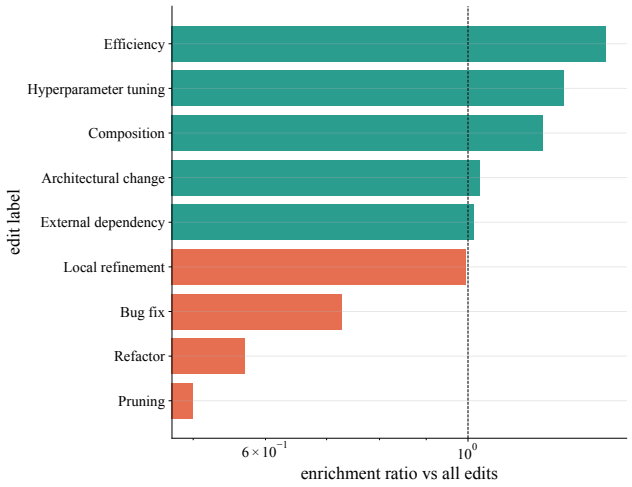


Figure 7: **Final-best-lineage enrichment (aggregate, robustness check).** Enrichment of each label along the lineage from each run's final best program back to the seed. *Efficiency*, *Hyperparameter tuning*, and *Composition* remain overrepresented relative to the all-edits base rate, supporting the best-so-far view in Figure 6.

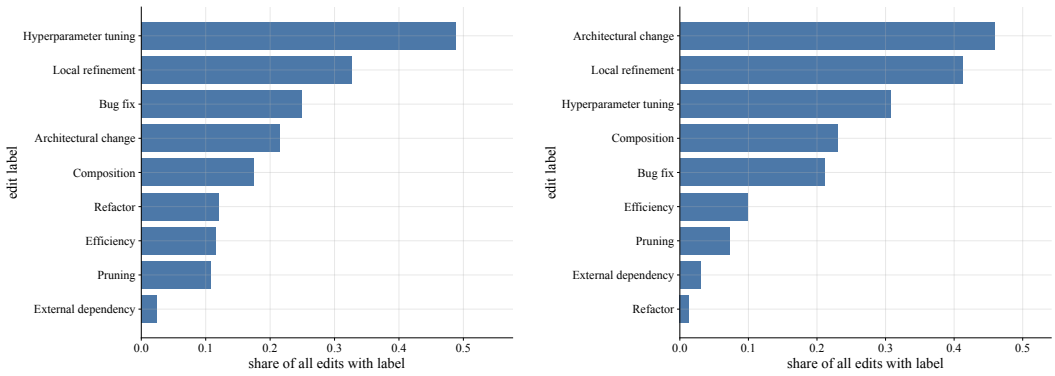
B.4 LLM-AS-JUDGE VALIDATION

Taxonomy origin. The 9-category edit taxonomy used in Section 5.5 was derived inductively from EvoTrace runs rather than imposed top-down. The first author sampled several classified runs and proposed an initial set of edit categories; these were discussed and refined with co-authors over multiple iterations on further sampled traces, with categories merged or split until the label set stabilised on the nine used to prompt the LLM judge: *hyperparameter_tuning*, *local_refinement*, *architectural_change*, *composition*, *efficiency*, *bug_fix*, *pruning*, *refactor*, and *external_dependency*. Table 10 gives a one-line working definition for each label; curated example diffs are provided in Appendix B.5.

Inter-rater reliability against the LLM judge. To validate the LLM-as-judge classifier we conducted a blind multi-label inter-rater reliability study on a stratified sample of 200 parent→child edits drawn from three classified runs. The first author labelled each edit without seeing the model's output; agreement was then computed against the `deepseek-chat` judge.

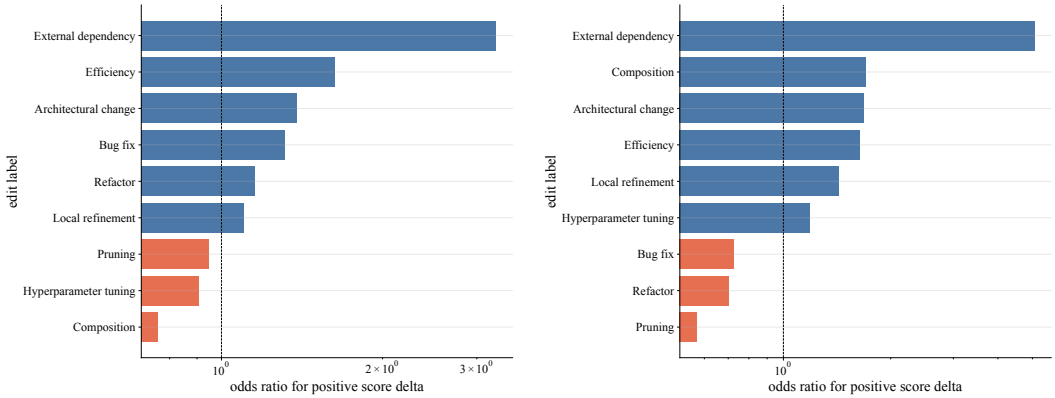
Across the nine categories we observed substantial overall agreement: macro Cohen's $\kappa = 0.77$, mean Jaccard $= 0.86$, micro-$F_1 = 0.90$, and exact-match accuracy $= 74.5\%$.

Ambiguity of external-dependency edits. The phrase "one failure case" in an earlier draft referred to label ambiguity, not to a failed evolutionary run. An edit that introduces JAX or Optax commonly also replaces a core algorithmic block, so it can correctly receive both *external dependency* and *architectural change*. The curated external-dependency example in Appendix B.5 is intentionally



(a) ALE (b) Math

Figure 8: **Edit-label prevalence by domain.** Frequency of each taxonomy label among labeled edits, split by domain. *Hyperparameter tuning* dominates in both domains, but *Composition* is more prominent on math while structural categories shift their relative weights between domains.



(a) ALE (b) Math

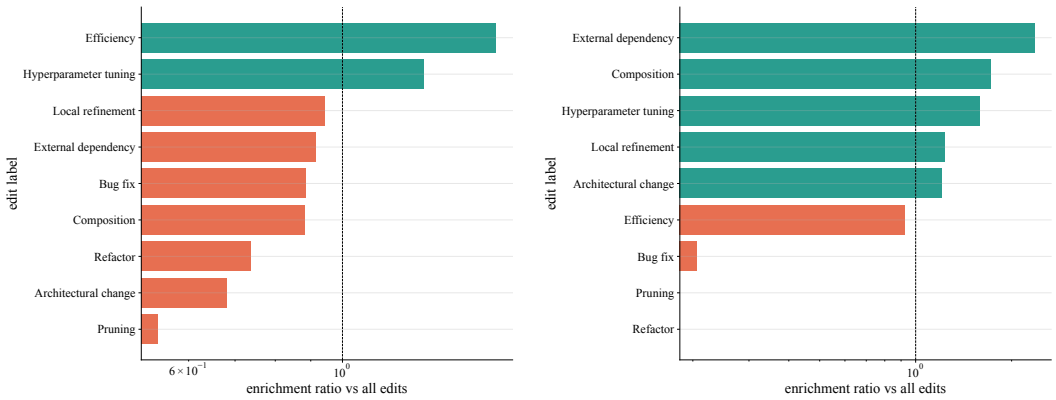
Figure 9: **Odds ratio that an edit beats its parent, by domain.** On ALE, *External dependency*, *Efficiency*, and *Architectural change* have the strongest positive associations. On math, *External dependency* is stronger and *Composition* plays a larger role than on ALE.

multi-label: it is the clearest visual example of a new library import, not an example in which the library label can be interpreted independently of the architectural change. We therefore retain all applicable labels and interpret label-level score results as associations rather than separate causal effects.

Library-hint intervention. Because 17 of 22 external-library edits in the math traces had positive score changes, we tested a prompt intervention that named task-relevant libraries without requiring their use and otherwise left the search setup unchanged. Averaged across GEPA, OpenEvolve, and EvoX, the mean best score improved on circle packing ($0.996204 \rightarrow 1.000373$), first autocorrelation ($0.992583 \rightarrow 0.997990$), and second autocorrelation ($1.013840 \rightarrow 1.015932$). This three-task result is a preliminary test of a hypothesis suggested by the taxonomy, not a causal estimate of the label itself or evidence for an edit-frequency quota.

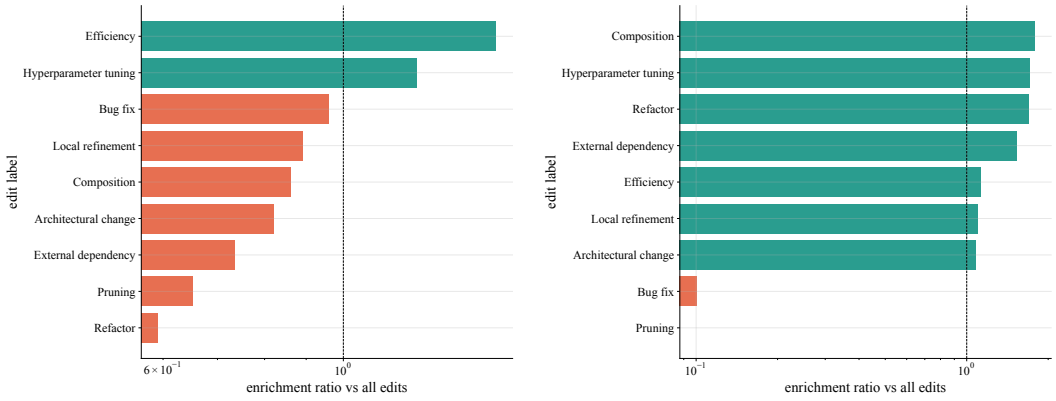
B.5 Curated examples per edit type

To make the taxonomy labels of Section 5.5 concrete, we illustrate each of the nine categories with one curated example diff drawn from the labeled corpus. Examples are selected for visual clarity rather than for the highest score delta; some carry more than one label (e.g., *Pruning*, *Composition*, *Bug fix*, *Efficiency*, and *External dependency* are each accompanied by other labels in the cleanest



(a) ALE (b) Math

Figure 10: **Best-so-far enrichment of edit labels by domain.** Enrichment of each label among best-so-far updates relative to the all-edits base rate, split by domain. The qualitative signal, a small set of categories (notably *Efficiency*, *External dependency*, and *Hyperparameter tuning*) overrepresented on successful intermediate steps, is consistent with the aggregate view in Figure 5, with domain-specific shifts in magnitude.



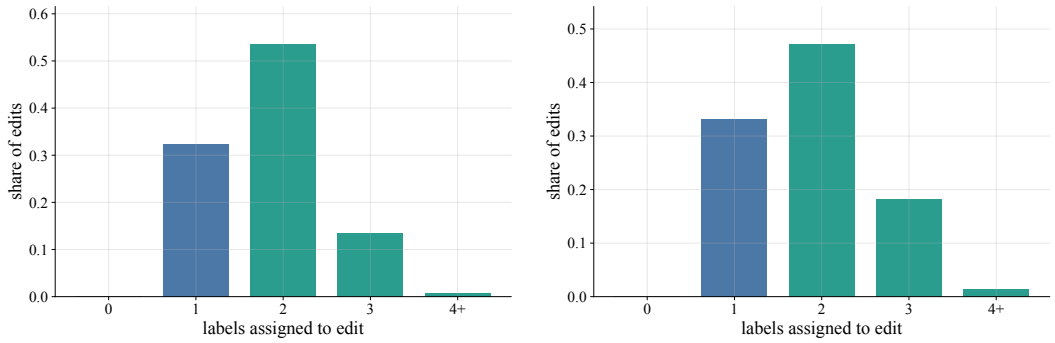
(a) ALE (b) Math

Figure 11: **Final-best-lineage enrichment by domain.** Robustness check for Figure 10, restricted to edits that lie on the lineage of each run's final best program. The enriched categories overlap heavily with the best-so-far view in both domains, with *Efficiency* and *Hyperparameter tuning* retaining their overrepresentation.

paper-ready example), which is itself part of the empirical story analyzed in Appendix B.3: most edits in the corpus are multi-label. Each example below lists run identity, iteration, label set, and score delta; full unified diffs and metadata are bundled with the dataset release.

Hyperparameter tuning. *Single cooling-rate change.* `openevolve_native / heilbronn_triangle`, iter 40, labels {hyperparameter_tuning}, $\Delta s = +0.0153$. The cleanest possible single-knob example: one numeric literal changes; the surrounding algorithm is unchanged.

```
@@ -74,7 +74,7 @@
     num_restarts = 25           # more restarts to escape local minima
     steps_per_run = 200000      # longer runs for better convergence
     T0 = 0.12                   # higher initial temperature
-    cooling_rate = 0.99992      # slightly slower cooling
+    cooling_rate = 0.99993      # slightly slower cooling (compensate more steps)
     best_min = 0.0
     best_points = None
```



(a) ALE (b) Math

Figure 12: **Distribution of labels per edit, by domain.** Most edits in both domains are multi-label: 52.4% of edits aggregate-wide carry exactly two labels and only 32.4% are single-label. The categories of Figure 5 should therefore be read as overlapping rather than mutually exclusive modes.

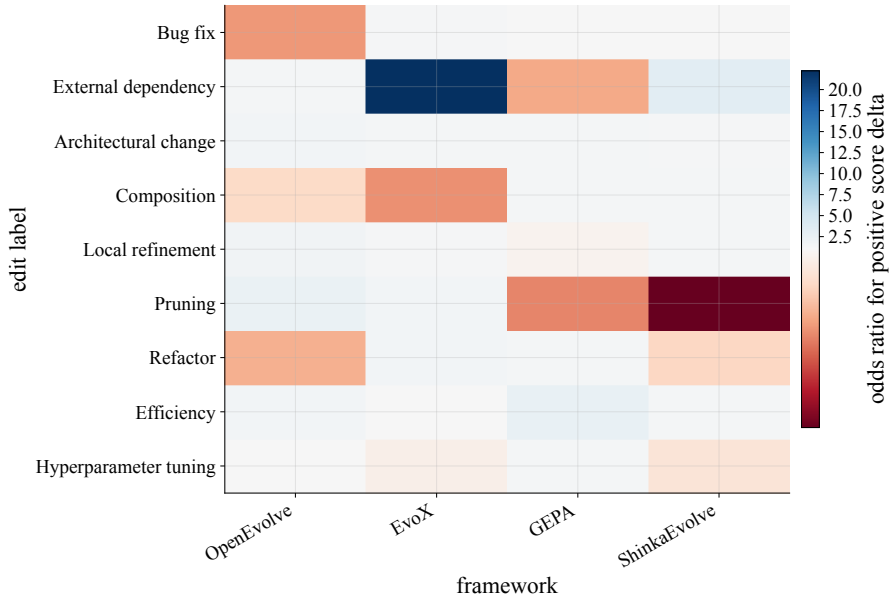


Figure 13: **Odds ratio that an edit beats its parent, by backend.** Some categories, notably *External dependency*, are consistently positive across backends, while others vary in magnitude. `openevolve_native` contributes only 2 runs to this corpus, so its column should be interpreted with a wider implicit confidence band.

Pruning. *Delete final global-shake phase.* `openevolve_native` / `heilbronn_triangle`, iter 38, labels {hyperparameter_tuning, pruning}, $\Delta s = +0.0670$. A whole final optimization phase is removed (not commented out or renamed); the diff also retunes several literals, so this is a pruning-plus-tuning compound example.

```
@@ -71,10 +71,10 @@
         return np.min(area)

     # Simulated Annealing parameters
-    num_restarts = 35           # increased restarts to escape local minima
-    steps_per_run = 250000      # longer runs for better convergence
-    T0 = 0.15                   # higher initial temperature for more exploration
-    cooling_rate = 0.99992      # slightly slower cooling
+    num_restarts = 25           # balanced restarts and run length
+    steps_per_run = 280000      # more steps for deeper exploration
+    T0 = 0.12                   # initial temperature
+    cooling_rate = 0.99993      # slightly slower cooling (compensate more steps)
```

Table 10: Working definitions of the nine edit categories. One curated example diff per category is given in Appendix B.5.

Label	Definition
Hyperparameter tuning	Change to one or more numeric literals or configuration values; control flow and surrounding algorithm unchanged.
Local refinement	Targeted edit within an existing routine; the routine's role and structure are preserved.
Architectural change	A core algorithmic block is replaced by a substantively different approach.
Composition	A new component (operator, phase, branch) is added alongside an existing one; existing logic is retained.
Efficiency	Same input–output behaviour reimplemented at lower asymptotic or constant cost (e.g. batching, vectorization, partial sorts, caching).
Bug fix	Correction of a latent defect such as a missing guard, wrong sign, off-by-one, or mishandled sentinel.
Pruning	A code path, phase, or feature is removed; the remaining program is kept intact.
Refactor	Behaviour-preserving restructuring: renaming, reordering, extracting helpers, or moving declarations.
External dependency	Introduction or removal of an import or external library.

```
     best_min = 0.0
     best_points = None

@@ -169,24 +169,6 @@
              best_min = current_min_ref
              best_points = points_ref.copy()

-    # Final global shake of best configuration to escape narrow local minima
-    if best_points is not None:
... [truncated, see full diff file] ...
```

Architectural change. *Brute-force candidate search replaced by closed-form selection.* `evox / ale_bench_ahc016`, iter 28, labels {architectural_change, local_refinement}, $\Delta s = +1.37 \times 10^7$. The program stops scanning many candidate graph sizes and switches to a derived closed-form rule.

```
@@ -203,40 +203,49 @@
     double epsilon_noise_rate;
     std::cin >> M_graphs >> epsilon_noise_rate;

-    double best_score = -1e100;
-    int best_N = 4;
-    Strategy best_strat = Strategy::GED;
-
-    // Evaluate GED strategies (N=4,5,6)
-    for (int Ncand : {4,5,6}) {
-        if (Ncand == 6 && M_graphs > 156) continue;
-        if (Ncand == 5 && M_graphs > 34) continue;
-        if (Ncand == 4 && M_graphs > 11) continue;
-        double score = estimate_ged_score(Ncand, M_graphs, epsilon_noise_rate);
-        if (score > best_score) { ... }
-    }
-    // Evaluate edge-count strategies (N=4..100)
-    for (int Ncand = 4; Ncand <= 100; ++Ncand) {
-        double score = estimate_edge_score(Ncand, M_graphs, epsilon_noise_rate);
-        if (score > best_score) { ... }
-    }
+    int N_for_GED_strat;
+    if (M_graphs <= 11) N_for_GED_strat = 4;
... [truncated, see full diff file] ...
```

Local refinement. *Add boundary bonus inside the existing scoring heuristic.* `evox / ale_bench_ahc015`, iter 94, labels {local_refinement}, $\Delta s = +4{,}295$. Small targeted change to a heuristic formula; surrounding algorithm intact.

```
@@ -283,6 +283,10 @@
                 if ((candy.c == 0 && min_target_c == 0) ||
```

```
                    (candy.c == GRID_SIZE - 1 && max_target_c == GRID_SIZE - 1)) {
                     bonus_val += PER_CANDY_BONUS_FACTOR;
+                }
+                // Additional bonus for being at the boundary of the target column range
+                if (candy.c == min_target_c || candy.c == max_target_c) {
+                    bonus_val += 0.5;
                 }
             }
         }
```

Composition. *Add swap mutation operator on top of existing plan search.* `evox` / `ale_bench_ahc026`, iter 57, labels {hyperparameter_tuning, composition}, $\Delta s = +9{,}252$. Main search intact; an additional mutation operator is layered on, alongside several constant retunings.

```
@@ -18,11 +18,11 @@
 // Constants for heuristic evaluation
-const double HEURISTIC_EMPTY_STACK_BONUS_SCORE = 1500.0;
+const double HEURISTIC_EMPTY_STACK_BONUS_SCORE = 1000.0;
 const double STACK_HEIGHT_PENALTY_FACTOR = 0.1;
 const int    HEURISTIC_LOOKAHEAD_WINDOW = 5;
-const double HEURISTIC_COVER_CRITICAL_PENALTY_PER_BOX_ABOVE = 4.0;
-const double HEURISTIC_MIN_LABEL_IN_DEST_FACTOR = 0.03;
+const double HEURISTIC_COVER_CRITICAL_PENALTY_PER_BOX_ABOVE = 5.0;
+const double HEURISTIC_MIN_LABEL_IN_DEST_FACTOR = 0.05;

@@ -473,7 +467,7 @@
         double op_choice_rand = RGen.a_double(0.0, 1.0);
-        if (op_choice_rand < 0.35 && N_CONST > 0) {
+        if (op_choice_rand < 0.25 && N_CONST > 0) {
... [truncated, see full diff file] ...
```

Bug fix. *Check strategy success and INF_COST sentinel.* `evox` / `ale_bench_ahc046`, iter 99, labels {bug_fix, hyperparameter_tuning}, $\Delta s = +56.2$. Parent silently ignored a helper's return value and invalid-cost sentinel; child explicitly guards against both failure modes.

```
@@ -378,7 +378,7 @@
-const int GREEDY_REOPTIMIZE_SUBSET_SIZE = 40;  // Balanced between exploration and speed
+const int GREEDY_REOPTIMIZE_SUBSET_SIZE = 170; // Full scan for best strategy per segment

@@ -798,12 +798,16 @@
             current_sa_choices[k] = current_best_strategy_code_for_k;

             SegmentExecResult final_segment_res_for_k_build;
-            apply_combined_strategy(current_best_strategy_code_for_k,
+            bool success_seg = apply_combined_strategy(
+                                    current_best_strategy_code_for_k,
                                     player_pos_sim_build,
                                     target_P_k,
                                     greedy_grid_sim_build,
                                     final_segment_res_for_k_build,
                                     true);
+            if (!success_seg || final_segment_res_for_k_build.turns == INF_COST) {
+                possible_greedy = false;
+                break;
+            }
```

Efficiency. *`std::sort` replaced by `std::nth_element`.* `evox` / `ale_bench_ahc027`, iter 46, labels {efficiency, hyperparameter_tuning}, $\Delta s \approx 9.2 \times 10^{18}$. Top-$k$ selection semantics are preserved; full sort is replaced by a partial-ordering primitive.

```
@@ -266,8 +266,6 @@
             TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.push_back(...);
         }
     }
-    std::sort(TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.begin(),
-              TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.end());

@@ -275,6 +273,13 @@
     // Stochastic: pick uniformly from the top 20% of cells
     int num_candidates = std::max(1,
         (int)TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.size() / 5);
+    // Use nth_element to partition: the first num_candidates elements are the largest
+    std::nth_element(TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.begin(),
+                     TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.begin() + num_candidates,
+                     TMP_CELL_DIRT_INFOS_LIST_GLOBAL_BUFFER.end(),
```

```
+                     [](const CellDirtInfo& a, const CellDirtInfo& b) {
+                         return a.weighted_dirt_contribution
+                              > b.weighted_dirt_contribution;
+                     });

@@ -600,7 +605,7 @@
-        const int POST_MAX_ATTEMPTS = 500;
+        const int POST_MAX_ATTEMPTS = 200;
```

External dependency. *Introduce JAX and Optax optimization pipeline.* `shinkaevolve` / `first_autocorr_ineq`, iter 94, labels {architectural_change, external_dependency}, $\Delta s = +0.991$. The strongest paper example of *external_dependency*: the imports of `jax` and `optax` are unambiguous and visually prominent. Note that the same diff also constitutes a large architectural change, illustrating that this label often appears in combination with others.

```
@@ -1,3 +1,127 @@
 # EVOLVE-BLOCK-START
-# Paste your original program here. Ensure the evolve block markers are present.
+import jax
+import jax.numpy as jnp
+import optax
+import numpy as np
+from dataclasses import dataclass
+
+@dataclass
+class Hyperparameters:
+    num_intervals: int = 600
+    learning_rate: float = 0.005
+    end_lr_factor: float = 1e-4
+    num_steps: int = 40000
+    warmup_steps: int = 2000
+
+class AutocorrelationOptimizer:
+    def __init__(self, hypers: Hyperparameters):
+        self.hypers = hypers
+        self.domain_width = 0.5
+        self.dx = self.domain_width / self.hypers.num_intervals
... [truncated, see full diff file] ...
```

Refactor. *Move `SimulationResult` definition before use.* `shinkaevolve` / `ale_bench_ahc026`, iter 68, labels {refactor}, $\Delta s = +9{,}248$. The cleanest refactor diff in the corpus: a type definition is moved to a more appropriate place without changing the algorithm or introducing new logic.

```
@@ -137,7 +137,13 @@
     }
 };

-// Forward declaration for helper functions
+// Define SimulationResult before any function that uses it
+struct SimulationResult {
+    long long energy_cost;
+    std::vector<std::pair<int, int>> ops_history;
+};
+
+// Forward declaration for helper functions (SimulationResult is now defined)
 std::pair<State, long long> simulate_up_to_k(
     const std::vector<std::vector<int>>& init,
     const std::vector<int>& plan,

@@ -319,11 +325,6 @@
     return run_simulation_from_intermediate_state(
         std::move(st), plan, 0, N, M, record_all);
 }

-struct SimulationResult {
-    long long energy_cost;
-    std::vector<std::pair<int, int>> ops_history;
-};
-
 int main() {
```

B.6 CYCLING: ADDITIONAL ANALYSES

The body section Section 5.1 reports the headline cycling result and the conservative intervention result. This appendix covers (i) a finer three-way classifier on the unified diff, (ii) context and whole-

program comparisons, (iii) hard suppression and two retrospective stopping thresholds, (iv) a model- and prompt-dependence breakdown of the tuning share, and (v) a null result on post-breakthrough cycling that did not survive cross-run aggregation.

Three-way classifier. Each parent–child edit is classified deterministically using three categories on the unified diff between $\mathbf{p}_t$ and p_t. *Literal recycling*: the added line is byte-identical to a line previously removed elsewhere in the lineage. *Tuning recycling*: the added line's number-collapsed skeleton matches a previously-removed line, but the numeric values differ (coefficient churn). *Trivial recycling*: comment-only or whitespace-only changes. The headline $\sim 30\%$ is the median across runs of the *literal-recycling* share. When all added lines are pooled rather than weighting each run equally, 31.9% are literal returns and a further 6.2% match after numeric values are replaced by placeholders, for a combined pooled share of 38.1%. We report the literal and tuning variants separately because only the first is byte-identical.

Context and whole-program returns. A returned line need not play the same role after the rest of the program changes. Among recycled-line events in the math runs for which both program contexts were retained, 57% occurred after the surrounding program had changed and 43% returned without such a context change. Exact returns to a complete earlier program are much rarer than line returns: 5.33% of candidates exactly match an ancestor program, falling to 0.84% after excluding one-step returns to the previous program. Thus the line-level rate should not be interpreted as a rate of behavioral or whole-program repetition.

Suppression and stopping interventions. We tested hard suppression in ten additional paired runs by preventing the search from reintroducing previously deleted lines. Relative to the corresponding unrestricted runs, nine pairs reached the same final-best score; one second-autocorrelation run worsened from 1.015833 to 0.994119. We therefore find no evidence that recycled lines should be blocked.

We next evaluated recycling as a retrospective stopping signal, applying it only after a run's best score had plateaued. A cumulative recycling threshold of 60% triggered in 10% of runs and saved 5.8% of evaluations overall; every triggered run stopped within 0.5% of its eventual final score. A 40% threshold triggered in 27.5% of runs and saved 18.0% of evaluations overall; only one run stopped more than 0.5% below its eventual final score, so 97.5% of all runs remained within 0.5%. These results support recycling as a possible stopping or handoff signal after stagnation, not as an edit to suppress.

Edit composition is model- and prompt-dependent. Restricting to code-changing lines, the median per-run share that the paired-skeleton classifier marks as *tuning recycling* is 8%, but the range is wide (2–44%). Holding the task fixed at `ahc015`, the tuning share varies sharply with the generator (Table 11). The diff-vs-no-diff axis is the strongest single predictor: the same model (`deepseek-reasoner`) drops from 20% to 2% tuning share when its diff-based generation is turned off.

Table 11: Tuning share of code-changing lines on `ahc015`, holding task fixed. The diff-vs-no-diff axis dominates model identity.

Generator	Tuning share
`gflash`	44%
`claude-haiku-4-5`	26%
`deepseek-reasoner` (diff)	20%
`deepseek-reasoner` (no diff)	2%

Negative result on post-breakthrough cycling. We initially hypothesized that cycling spikes immediately after a best-so-far event (a refractory period in which the search churns over surrounding code). The change in cycling rate in the 5 iterations after each best-so-far event has mean -0.005, median -0.021, and range $[-0.39, +0.23]$ across 26 runs. Some runs show large positive spikes, others large drops. The hypothesis does not survive cross-run aggregation and we report it as a null.

B.7 PUBLIC-VS-PRIVATE GENERALIZATION ON ALE

ALE-bench public scores are not the held-out judging metric. We re-score every ALE run's public best-so-far chain on the private test set used by AtCoder (n=30 run/problem pairs across the four main backends, covering 10 ALE problems). Two of the four frameworks overfit on at least 30% of the problems they were scored on, and the same problem can flip generalization sign between frameworks (Table 12). On `ahc024`, `openevolve` found a +1,606 rating-point private gain (aligned), while `shinkaevolve`, on the same problem, lost 1,610 rating points despite a positive public score change. On `ahc027`, three of four frameworks (`evox`, `gepa`, `openevolve`) overfit; only `shinkaevolve` generalized. Per-framework counts are summarized in Table 13: `evox` aligned 0 of 8 scored runs, `gepa` 1 of 7, `openevolve` 4 of 9, `shinkaevolve` 4 of 6. The public best-so-far chain is therefore unreliable as a single-number summary of an ALE run; in our data, problem identity is a stronger predictor of overfit than framework identity.

Table 12: **Public vs. private generalization across frameworks on ALE.** Each cell reports the change in AtCoder rating-point performance from seed to the final public-best program along the run's public best-so-far chain, when re-scored on the held-out private test set. Bold cells flag *overfitting* (public ↑, private ↓); "—" indicates a single-event lineage, an unscorable seed, or a missing private metric.

problem	evox	gepa	openevolve	shinka
ahc008	+0	+0	+37	+38
ahc011	−3	—	—	—
ahc015	+0	+0	+0	+2,194
ahc016	−1	+5	+2	+7
ahc024	+0	—	+1,606	−1,610
ahc025	+0	+0	−16	—
ahc026	+0	+0	+1,735	+10
ahc027	−45	−489	−69	+30
ahc039	+0	+0	+0	−32
ahc046	+0	+0	−1,995	+0

Table 13: **Per-framework counts of aligned vs. overfitting public→private trajectories on ALE.** A run is *aligned* if both public and private scores improved from seed to final, *mild overfit* if private worsened by $\leq$ 200 rating points despite a public gain, and *severe overfit* if by more than 200 points; the remainder is no movement on private.

framework	runs scored	aligned	overfit (mild)	overfit (severe)
evox	8	0	3	0
gepa	7	1	0	1
openevolve	9	4	2	1
shinka	6	4	1	1

B.8 A WALK-THROUGH OF ONE REPAIR CYCLE

Figure 3 traces one cycle in an EvoX `ahc008` run. The iteration-1 child changes a wall-building heuristic, including replacing `STAND_OUTSIDE_INNER_SAFE_PENALTY = 1000` with `... = 10000`; its score falls from 3,669,080.6 to 2,426,100.9. At iteration 4, a descendant of that child reverses much of the edit. Of its 22 added lines, 20 are byte-identical to lines deleted at iteration 1, and the score returns to 3,669,080.6. The edit is therefore locally successful but makes no run-level advance: it repairs the branch by recovering the earlier program behavior.

B.9 BAYESIAN OPTIMIZATION BASELINE

This section documents the BO baseline used for the tuning-gap analysis in Section 5.3. The goal is to measure $\hat{f}_{\mathrm{BO},24}(s_0)$ for a fixed seed structure s_0 by tuning only its embedded numeric constants, with

no further structural search. The pipeline has three stages: (i) a single LLM call that identifies tunable knobs and proposes intervals, (ii) a deterministic rewrite that exposes those knobs as a top-level parameter block, and (iii) a short `gp_minimize` run that calls the original evaluator harness.

Knob identification (one LLM call). We send the program source to `deepseek-reasoner` via an OpenAI-compatible chat endpoint and ask for a JSON list of candidate knobs. There is no agentic loop and no retry: a single structured-output call returns a list of objects with fields `name`, `source_literal`, `context_line`, `default`, `low`, `high`, `scale` (`linear` or `log`), `kind` (`int` or `float`), and a one-sentence `rationale`. The system prompt restricts candidates to solver tolerances, iteration/sample budgets, step sizes and learning rates, soft penalty/reward weights, cooling rates, and threshold dispatches; problem constants (population sizes that the evaluator reads, grid dimensions, the n in n-circle packing), mathematical identities ($\sqrt{2}$, π), array shapes, and boolean flags are explicitly excluded. Range guidelines are calibrated by parameter family (log-scale spans of 10^2 for step sizes and weights, 10^3 for tolerances; linear $[0, 1]$ for probabilities and acceptance ratios; etc.), with a hard rule that any range with $\mathrm{high}/\mathrm{low} \geq 100$ uses a log scale. The prompt caps the proposal at 8 knobs, since BO budget scales poorly past that, and instructs the model to skip any literal that appears multiple times in the source (ambiguous replacement target).

Validation and rewrite. Each returned spec is validated: the `source_literal` must appear byte-identically in the source, and the `context_line` must match a line in the file. For Python targets, a `PARAMS = {...}` dict is injected at the top of the file (after shebang/encoding/docstring) and each accepted literal is replaced in its context line by `PARAMS["name"]`. For C++ targets, a block of `#define _BO_NAME value` macros is inserted after the last `#include` and the literal is replaced by the corresponding macro. Replacement uses a regex that excludes longer-number neighbors (so `0.1` does not match inside `0.123`). Specs whose literal can be found in the source but not in the proposed context line are dropped without retry; the BO then runs over the surviving knobs only, so the effective per-target knob count (median 6 in the runs reported in Section 5.3) is generally smaller than the LLM's nominal proposal.

BO loop. Tuning runs use `skopt.gp_minimize` (scikit-optimize) with 24 evaluator calls per target: 8 random initial points followed by 16 BO acquisitions over a Gaussian-process surrogate. Each call substitutes the current parameter vector into the rewritten source (regenerating the `PARAMS` dict or rewriting the `#define` block) and invokes the same evaluator harness used during the original evolutionary run, so $\hat{f}_{\mathrm{BO},24}(s_0)$ and $f^{\star}_{\mathrm{evo}}$ use the same task evaluator. Reported numbers in Section 5.3 (Table 3) are best-of-24 minus the seed program's score $f(p_0)$.

Table 14: **Finite-budget BO gain by search stage.** Programs were selected by their position in the evolutionary run, before observing the BO result.

Starting program	Median relative gain
Early best, through iteration 30	$+4.64\%$
Selected intermediate program	$+6.08\%$
Final-best program	$+0.14\%$

Edit order along winning lineages. The first half of a final-best lineage is 52% structural edits and 31% tuning edits, compared with 35% structural and 43% tuning in the second half. The positional test is marginal ($p = 0.055$), and the direction holds in 10 of 15 runs in a within-run sign test ($p = 0.30$). We therefore treat the pattern as exploratory rather than evidence for a general search schedule.

Knob-identification prompt. The system prompt sent to `deepseek-reasoner` is reproduced verbatim below; the user message is just the program source wrapped in a fenced block tagged with the language.

```
You are an expert at identifying tunable hyperparameters in heuristic
optimisation code.
```

```
Your task: read a program and return a JSON list of numeric constants
that are good candidates for Bayesian-optimisation tuning, WITH
SENSIBLE RANGES.

GOOD candidates:
  - Solver tolerances / convergence thresholds (e.g. 1e-6, 0.001)
  - Iteration / restart / sample budgets (e.g. 1000, 50, 10)
  - Step sizes, perturbation scales, learning rates
  - Soft penalty / reward weights (e.g. 0.5, 2.0)
  - Cooling / annealing rates (e.g. 0.95, 0.99)
  - Threshold dispatches (e.g. `if depth < 30: return X else return Y`)

BAD candidates (do NOT include):
  - Problem constants (n=26, n_circles=26, dim=2, GRID_SIZE=10)
  - Mathematical identities (sqrt(2), pi, e)
  - Array shapes / index bounds
  - Evaluator-facing constants (timeout values that the harness reads)
  - Constants inside identities the model should preserve
    (vertices, basis vectors)
  - True boolean flags (use=True)

For EACH selected knob, return:
  - name           : a unique snake_case identifier (e.g. "step_size")
  - source_literal : the EXACT numeric literal as it appears in the
                     source (e.g. "0.1", "1e-6", "0.95"). Must be
                     byte-identical.
  - context_line   : the line of code where the literal appears
                     (verbatim, used as a disambiguator for replacement)
  - default        : the default value (= source_literal as a number)
  - low / high     : interval bounds for BO (see RANGE GUIDELINES)
  - scale          : "linear" or "log" (see RANGE GUIDELINES)
  - kind           : "int" or "float"
  - rationale      : one short sentence

RANGE GUIDELINES -- cover orders of magnitude when the parameter family
warrants it. Don't propose timid +-20% intervals: BO can only discover
what the search space contains. Use these defaults:

  parameter family            scale   range relative to default
  --------------------------  ------  ---------------------------------
  learning rate / step size   log     [default/100, default*100]
  temperature / amplitude     log     [default/100, default*100]
  tolerance / threshold       log     [default/1000, default*1000]
  penalty / reward weight     log     [default/100, default*100]
  cooling factor (0<x<1)      linear  [0.5, 0.99999]   (always wide)
  acceptance prob / sigmoid   linear  [0.0, 1.0]
  restart count / pop size    log     [max(1, default/10), default*10]
  iteration budget            log     [default/5, default*5]
  sample count                log     [max(1, default/10), default*10]
  small-int categorical       linear  [1, max(default*5, 20)]

If a parameter doesn't fit a family above, default to:
  - log scale if default is a positive non-integer < 1 or > 100
  - linear scale otherwise
  - range that spans at least one order of magnitude (high/low >= 10)

HARD RULES:
  - If high/low >= 100, scale MUST be "log" (BO converges much faster
    on log-scale priors over wide intervals).
  - Cooling factors and probabilities stay within their natural [0, 1]
    range even if that constrains low/high.
  - For integers: low = max(1, floor(low_proposed));
                  high = ceil(high_proposed).

CONSERVATIVE RULES:
```

```
  - Pick at most 8 knobs. Fewer is fine -- quality over quantity.
  - If a literal appears multiple times in the source, skip it
    (ambiguous).
  - If unsure whether a constant is tunable, skip it.

Return ONLY a JSON object of the form:
    {"hparams": [ { ...one knob... }, ... ]}
No commentary, no markdown fences.
```