OPINIONATED OPTIMIZERS: WHY AGENTS STRUGGLE TO IMPROVE AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Meta-optimization systems are increasingly used to improve agent harnesses for a given task through an iterative loop in which a meta-agent *proposes* edits, *evaluates* the resulting harnesses, and *selects* new candidates to build on. Prior work has shown that meta-optimized agents often fail to generalize beyond the data seen during optimization, but the reasons for this remain unclear. We argue that this overfitting occurs because the proposer meta-agent is *opinionated*: updates reflect prior preferences for particular strategies, unlike the analytically derived updates of gradient-based learning. We test this account through a systematic analysis of over 3 million recorded agent rollouts, organized around 216 optimization runs spanning 4 frontier methods and 3 benchmarks. To do so, we develop the Meta-Optimization Testbed (**MOT**), a standardized framework for studying meta-optimization in a controlled setting. Our analysis identifies 9 recurring failure modes spanning 3 stages of the meta-optimization loop: (a) *proposals* draw from a narrow set of familiar strategies shaped more by the proposer's priors than by the task, (b) noisy *evaluations* provide weak evidence for revising those priors, and (c) *selection* retains local improvements under method-specific rules, but further search rarely compounds them into additional gains. By tying failures to specific stages of the meta-optimization process, our analysis provides a clear roadmap for future research; we release **MOT** and the annotated traces to support it.

1 INTRODUCTION

As LLM agents become more capable, they are increasingly being used for *meta-optimization*, wherein a proposer meta-agent suggests edits to a worker agent's harness to improve its performance on a dataset, based on feedback from runs performed on a specified search subset. The resulting candidates are then iteratively evaluated, selected, and used as the basis for further edits. Meta-optimization methods such as GEPA (Agrawal et al., 2025), AdaEvolve (Cemri et al., 2026), Meta-Harness (Lee et al., 2026b), and CRO (Yu et al., 2026a) have reported strong performance on diverse datasets. Yet, prior work also finds that these methods often overfit the data encountered during optimization (Zhang et al., 2026b; Fu et al., 2026; Wang et al., 2026). This begs the question: why do capable agents still struggle to reliably optimize other agents?

We argue that the answer lies in how meta-optimization differs from conventional gradient-based optimization. Rather than following an analytically derived update rule, meta-optimization uses the inductive biases of an *opinionated* proposer meta-agent to author updates. The proposer goes into the meta-optimization process with existing priors about effective strategies, interprets execution feedback through the lens of these priors, and selects strategies for subsequent edits. This gives meta-optimization unusual flexibility, allowing it to optimize arbitrary prompts, programs, and workflows from sparse, qualitative feedback, but also gives rise to new failure modes, which we analyze next.

We conduct a large-scale empirical analysis of more than 3 million rollouts, organized around a 216-run main-grid design spanning 4 frontier meta-optimization methods (GEPA, AdaEvolve, Meta-Harness, and CRO) and 3 benchmarks (GPQA-Diamond (Rein et al., 2023), CharXiv (Wang et al., 2024), and τ^2-bench (Barres et al., 2025)), with Sol and Opus proposers and a Luna worker; a separate side grid studies Haiku and gpt-oss-120B workers. Analysis-specific coverage is summarized in Table 2. To study these in a controlled setting, we introduce the Meta-Optimization Testbed (**MOT**). **MOT** standardizes evaluation and records candidate harnesses, execution traces, per-example out-

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.

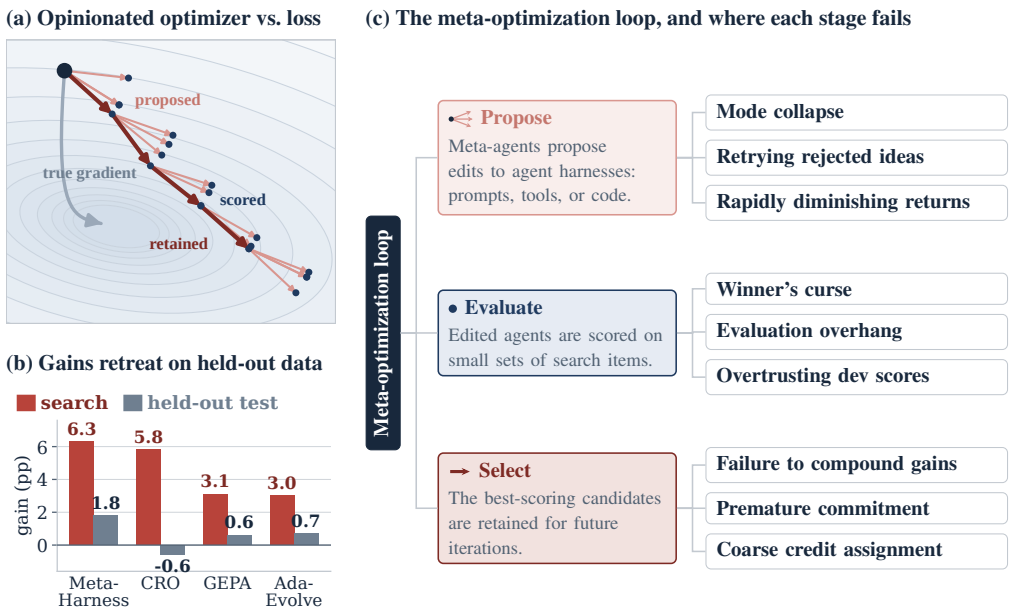

Figure 1: **Pitfalls of an opinionated optimizer.** (a) A schematic run of opinionated optimization on a loss surface. From the start (dark dot), the true gradient (grey) bends into the basin. At each step it proposes the changes (pink arrows), scores each one once (navy dots), and retains the one whose score looked best (red arrows), without seeing the slope. (b) An example of this pattern in our study: the mean gain of the final agent over its starting agent on the search set (red) and on held-out test set (grey). (c) The three stages of the meta-optimization loop and our failure mode taxonomy with definitions in Table 19.

comes, ancestry, and selection decisions using Shepherd (Yu et al., 2026a), enabling post-hoc reconstruction of a run, even without modifying method code. It also includes a pipeline for decomposing proposer edits into *interventions* or conceptually coherent and distinct strategies, such as "majority voting". By tracking them, we can trace how the proposer's priors evolve over the course of a run.

Our analysis reveals 9 recurring failure modes in how the meta-optimizer forms, updates, and acts on beliefs during search (see fig. 1). First, **proposer beliefs are mode-collapsed and have inertia**: proposers repeatedly draw on a narrow repertoire of familiar strategies, relying on prior beliefs rather than adapting to execution feedback. Second, **evaluation provides weak signal for updating beliefs**: candidates are evaluated on small development sets to provide generalization signal, where noise from stochastic worker executions can dominate. Third, **further search rarely compounds improvements**: optimizers often continue editing improved candidates, but this follow-up seldom yields another retained gain. Method-specific acceptance rules shape which local improvements survive.

We further examine how these failures vary across proposer models, worker models, and meta-optimization methods, and show that they have direct implications for system design. To further validate our account of the failures, we combine proposer edit effects and noisy selection in a statistical model, which reproduces reported gains across 38 held-out Meta-Harness runs with a mean absolute error of 2.51 points. Its agreement with search gains does not establish transfer to test examples (§6.4).

In summary, we contribute:

- **A systematic analysis of meta-optimization.** We identify 9 recurring failure modes, showing that proposals remain anchored to familiar strategies from the proposer's prior beliefs, evaluations provide unreliable evidence of progress for updating those beliefs, and early improvements are rarely built upon to provide further gains;
- **A testbed for studying meta-optimization.** We will release **MOT**, the first standardized testbed for studying meta-optimization methods under shared evaluation controls;

- **A dataset of annotated meta-optimization runs.** We will release **MOT-Trace**, containing 20,392 labelled edits and 11,350 intervention records from 212 runs (202 Luna and ten weaker-worker runs).

2 Related Work

Meta-optimization. Harness search spans prompt optimization (Yang et al., 2023; Khattab et al., 2024; Yuksekgonul et al., 2024; Agrawal et al., 2025), reflective agents (Shinn et al., 2023), workflow and program evolution (Hu et al., 2024; Zhang et al., 2025b; Novikov et al., 2025; Yu et al., 2026a), and self-modifying coding agents (Zelikman et al., 2024; Zhang et al., 2025a; Lee et al., 2026b;a). These approaches differ in their editable objects and search policies but share proposal, evaluation, and selection.

Limits to improvement. Proposer concentration (Jiang et al., 2025; Zhang et al., 2025c), unreliable self-correction (Huang et al., 2024; Song et al., 2025), selection optimism and overfitting (Smith & Winkler, 2006; Dwork et al., 2015; Recht et al., 2019; Zhang et al., 2026b), and accumulated drift (Yu et al., 2026b; Zhang et al., 2026a; Wang et al., 2026) have been found to challenge this meta-optimization loop. Remedies involve improving failure feedback (Chen et al., 2026), selection (Toledo et al., 2025; Xia et al., 2026), or evaluation efficiency (Nian et al., 2026; Du et al., 2026).

Evaluation and diagnosis. Existing meta-optimization benchmarks assess harness editing, transfer, and compounding (Ursekar et al., 2026b;a; Gao et al., 2026; Chi et al., 2026; Lin et al., 2026). In contrast, we compare methods under shared controls, track interventions across the stages of the meta-optimization loop, and test a model of reported progress on held-out runs.

3 Preliminaries

Meta-optimization aims to improve a *worker agent*, consisting of a fixed base LLM w and an editable *harness* h containing the prompts, tools, memory, and code that control its calls (Lee et al., 2026b), on a task dataset $\mathcal{D}$ partitioned into training, search, and test sets: $\mathcal{D}_{\text{train}}$, $\mathcal{D}_{\text{search}}$, and $\mathcal{D}_{\text{test}}$. A *meta-optimization system* M consists of a *proposer* agent p together with a harness for running the following process: starting from a seed harness h_0, M iteratively proposes candidate edits, evaluates the resulting harnesses, and selects which candidates to retain or build on, while keeping w fixed. p may receive labels and execution traces from $\mathcal{D}_{\text{train}}$, while $\mathcal{D}_{\text{search}}$ provides aggregate scores but no labels or traces. $\mathcal{D}_{\text{test}}$ is withheld until final evaluation. Table 1 summarizes the meta-optimization systems analyzed in this paper in these terms.

Table 1: **How meta-optimization methods work.** GEPA edits prompts; the others can edit full harnesses. Full rules: Appendix Table 3.

	GEPA	AdaEvolve	Meta-Harness	CRO
Proposer sees	minibatch traces	evaluation scores	all training traces	all training traces
Evaluated on	search set	training set	search set	search set
Selected by	beating parent	outranking similar programs	decided by proposer	decided by proposer

Runs, candidates, and edits. A *run* applies M starting from h_0 and produces a sequence of *candidate* harnesses. In a single meta-optimization step, M may perform multiple *edits*: each *edit* evolves a *parent* harness to produce a *child* harness. A run typically returns the best-performing candidate on the search set as its optimized worker harness.

From edits to interventions. A single edit may implement several strategies: for example, introducing both majority voting and self-consistency. We thus decompose edits into **interventions**: coherent functional changes implementing a single strategy. An edit *introduces* an intervention when it first appears in the run, *develops* it when it is carried over from the parent, *removes* it when it is present in the parent but absent from the child, and *reintroduces* it when it appeared earlier in the run but is absent from the current parent. Tracking interventions therefore lets us track how the proposer's beliefs evolve over the course of a run.

Scores and generalization. For a fixed set of examples D, let $S_D(h)$ denote the expected score of harness h and $\hat{S}_D(h)$ an observed evaluation. Because worker executions are stochastic, repeated evaluations of the same harness on the same examples can yield different values of $\hat{S}_D(h)$; we refer to this variation as *evaluation noise*.

4 META-OPTIMIZATION TESTBED

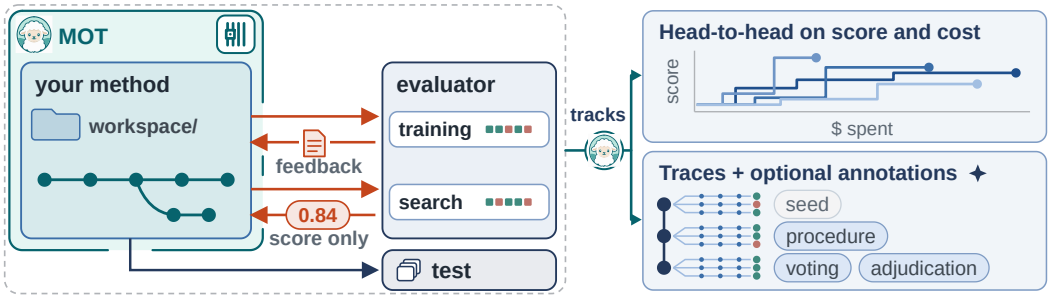

Figure 2: **The Meta-Optimization Testbed (MOT).** MOT standardizes task splits, scoring, and cost accounting while preserving method-specific search. Built on Shepherd, its versioned run records support performance comparisons and optional intervention annotation.

Analyzing meta-optimization requires access to the full optimization trajectories. However, reconstructing these trajectories after a run is difficult because the state required is distributed across model calls, tool use, filesystem changes, evaluations, and search decisions. Consequently, this analysis typically requires method-specific instrumentation. We therefore introduce the *Meta-Optimization Testbed* (**MOT**), a standardized framework that makes these trajectories observable and reproducible without needing to modify any method code (Figure 2).

Implementation. **MOT** is built on Shepherd (Yu et al., 2026a), a runtime that automatically captures model logs and workspace state resulting from agentic execution within its sandboxed environments as Git-like traces. This lets us record proposer and worker agent trajectories without modifying the vendored implementations of studied methods, while also isolating optimizer-controlled state from the surrounding evaluation infrastructure. **MOT** builds on this runtime to provide shared tasks and scorers, controls over the execution feedback exposed to the proposer, cost accounting, and versioned candidate histories that track lineage, scores, and selection decisions. Search and test data remain outside the run's sandbox (see Appendix B for isolation coverage and audits).

Extending MOT. **MOT** separates meta-optimization methods from benchmark-specific execution through lightweight, easily implemented adapters. A method adapter connects an optimizer's native search loop to common execution and evaluation interfaces, while a task adapter defines data splits, the execution harness, and the scorer (current inventory: Appendix B.1). The framework also supports Terminal-Bench-2 through Harbor, including editable terminal agents, task-verifier scoring, trajectory capture, and cost accounting.

Intervention-level analysis. **MOT** can also automatically label the conceptually coherent strategies or *interventions* used by the proposer agent during a run. Claude Sonnet is used for these annotations, by comparing each edit with the run's intervention registry at any point during the run (§3). A second pass assigns shared strategy labels (*task codes*) across runs. To validate this annotation pipeline, we carry out human annotations with 10 independent PhD students. They audit 180 annotations, with 2 reviewers per case. For task codes, the model matches 41 of 56 cases with clear human consensus (82.6% sampling-weighted agreement), while 66 of the 120 intervention cases require adjudication on at least one criterion (more details in Appendix D.2). For each edit, we additionally record whether the optimizer retained it and whether its score exceeded that of its parent by at least one estimated standard error (Appendix G).

5 EXPERIMENTS

5.1 SETUP

We evaluate four meta-optimization methods: GEPA, AdaEvolve, Meta-Harness, and CRO, on three tasks: GPQA-Diamond, τ^2-bench, and CharXiv. For each task, we construct three starting agents of increasing strength, which we call *minimal*, *medium*, and *strong* (more details in Appendix B). Search and test evaluations are reported as an aggregate of 3 executions. We report results with 2 proposer models (Claude Opus 5 and GPT-5.6-Sol) and 3 random seeds, yielding a total of 216 runs with GPT-5.6-Luna as the worker model. We also do ablations replacing Luna with Claude Haiku 4.5 or gpt-oss-120B while fixing Opus 5 as the proposer, adding 72 runs. Each run has a $150 budget and a 72-hour time limit.

Search and test gain. We define *search gain* as the largest improvement over the starting agent observed on the search set during a run, while *test gain* is the improvement of the returned agent over the starting agent on the held-out test set. For every method, including CRO, test gain is measured against the pooled score of the original starting agent for that task and starting tier; CRO's internal conversion does not change the reference. Appendix C gives the precise definitions and method-specific details; Appendix Table 6 reports the nine pooled baselines and their measurement counts.

5.2 MAIN RESULTS

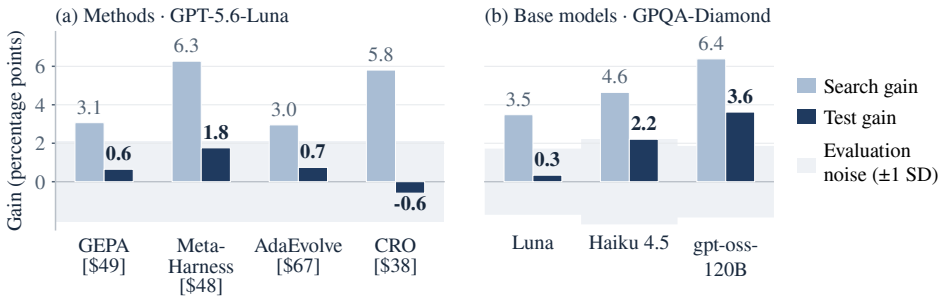

Figure 3: **Search gains largely disappear on held-out evaluation.** Mean gains by (a) method with a Luna worker and (b) worker on GPQA. Shading shows single-evaluation noise (± 1 SD); brackets show mean run cost.

Search gains largely disappear on held-out evaluation. Figure 3 shows our results. All 4 methods improve search-set performance on average, but only a small fraction of these gains survives on held-out examples. With GPT-5.6-Luna as the worker, GEPA improves by 3.1 points on search but 0.6 points on test; Meta-Harness by 6.3 versus 1.8 points; AdaEvolve by 3.0 versus 0.7 points; and CRO by 5.8 points on search while decreasing test performance by 0.6 points.

The same gap appears across worker models on GPQA-Diamond, although the weaker workers retain more of their search gains in the current subset. Haiku 4.5 improves by 4.6 points on search and 2.2 on test, while gpt-oss-120B improves by 6.4 and 3.6 points, respectively, compared with 3.5 and 0.3 points for Luna. We also evaluate off-the-shelf coding-agent baselines under the same held-out evaluation. Running Luna directly through Codex CLI performs similarly to the minimal starting agent, while coding agents allowed to construct a harness from an empty program achieve stronger performance in single exploratory runs. Appendix C.1 reports the results for these runs. Next, we analyze the failure modes behind this overfitting.

6 FAILURE MODES OF META-OPTIMIZATION

Meta-optimization involves an *opinionated* proposer starting with prior beliefs about effective strategies, interpreting execution feedback to update those beliefs, and proposing changes to the worker harness based on the resulting beliefs. We examine how these beliefs are expressed by tracking how

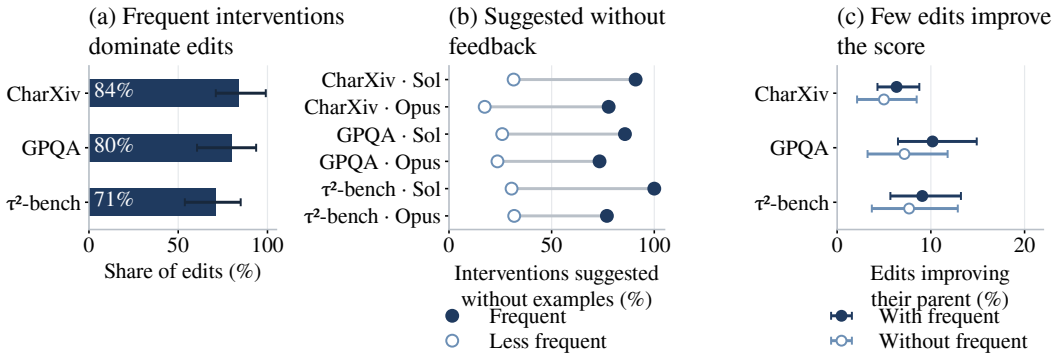

Figure 4: **Familiar interventions dominate proposals but rarely yield measured gains.** *Frequent interventions* appear in at least half of a task's runs. (a) Median edit share using them. (b) Share of frequent and less frequent categories suggested without feedback; frequency uses each proposer's own runs. (c) Mean share of edits improving their parent by at least one estimated standard error. Error bars: interquartile ranges in (a), 95% run-bootstrap intervals in (c). Appendix F.

interventions evolve, what evidence evaluations provide about their benefit, and which candidates become the basis for future edits (Figure 1). These intervention annotations create **MOT-Trace**, a dataset of 20,392 labelled edits and 11,350 intervention records from our 212 runs. (Appendix D.1). Appendix Table 19 summarizes the nine failure modes we identify, which we split across the proposal, evaluation and selection stages of meta-optimization and describe in detail below.

6.1 Proposal Stage

Search concentrates on a small set of familiar interventions. Proposers repeatedly draw on familiar strategies based on their prior, *mode collapsed* beliefs, resulting in universal interventions such as step-by-step chart reading, copying labels verbatim, and explicit arithmetic. We observe that proposers continue returning to these interventions even when edits involving them rarely improve performance. To measure this concentration, we call an intervention *frequent* if it appears in at least half of a task's runs. We identify 8–14 intervention categories that meet this threshold for each dataset. Proposers reuse them throughout search: a median of 71–84% of each run's edits contains at least one frequent intervention across tasks (Figure 4a). Despite their frequent reuse, edits involving these interventions rarely improve performance (Figure 4c). We count an improvement when the edit's score exceeds its parent's by at least one estimated standard error (§4; details in Appendix F).

Frequent interventions arise from proposer priors We give each proposer model the starting code and task description and ask it to suggest improvements, without task instances, scores, or execution traces. For each model, we divide the interventions observed in its optimization runs into frequent and *less frequent*. We then measure what percentage of each group it suggests without any execution feedback. Both proposer models recover a larger share of frequent interventions in all six task–model comparisons (Figure 4b). This supports interpreting many frequent interventions as *prior-driven*: they reflect strategies the proposer can suggest before observing failures. We next examine whether the failures shown during search support their use.

Does execution feedback support the proposed intervention? We compare GEPA steps where an intervention is first added with ablations where the intervention remains absent. An LLM judge (Claude Sonnet 5) assesses whether the intervention could plausibly address a shown failure. It receives the intervention description, current instruction, examples, and worker outputs, but sees neither the proposed edit nor whether the step is an introduction or a control. On GPQA, frequent interventions are often judged plausible even at control steps where GEPA does not propose them. Less frequent interventions receive support more selectively at introduction steps. An intervention can therefore seem broadly plausible without the shown failures explaining why it is proposed at that step. (full comparisons in Appendix F.3).

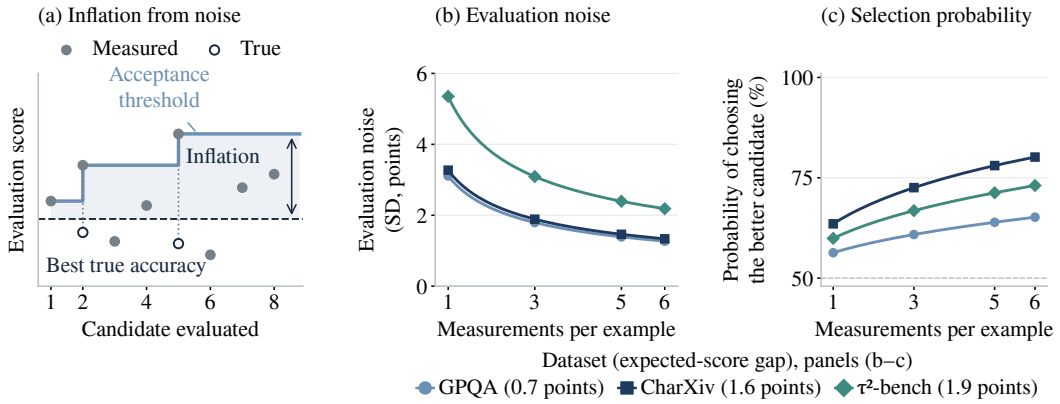

Figure 5: **More measurements reduce noise, but close candidates remain difficult to distinguish.** (a) Selecting high measured scores can raise the acceptance threshold without improving true accuracy. Filled dots show measured scores; hollow dots mark two candidates' true accuracies. Dotted connectors show measurement errors, and shading marks inflation above the best true accuracy. (b) Evaluation noise after averaging repeated measurements. (c) Probability of choosing the better of *two candidates*, each measured equally often, by comparing their average scores. The *expected-score gap* is how much better one candidate is expected to perform before evaluation noise. We estimate a typical gap for each dataset from variation between proposals and combine it with that dataset's measured noise in a Gaussian model (Appendix G.1).

6.2 Evaluation Stage

Winner's curse: apparent progress often comes from favorable noise. Evaluation should test whether a proposed edit improves the worker harness, but repeated measurements of the same candidate can yield different scores. The *winner's curse* (Smith & Winkler, 2006) shows that selection by the highest score favors candidates that are favored by noise. As search evaluates more candidates, its best recorded score can thus rise even without finding a better agent (Figure 5a). When that score becomes the threshold for accepting later candidates, the favorable measurement error carries forward as apparent evidence that the retained harness is better. Across 150 replacements of the best-so-far agent, the new candidate scores an average of 1.23 points higher on search but only 0.30 points higher on test. Promotion curves and the related failure modes *evaluation overhang* and *overtrusting dev scores* are provided in Appendix G.

Small gains are expensive to distinguish. Averaging k independent measurements per example reduces evaluation noise as $1/\sqrt{k}$: halving the noise requires four times as many measurements. What matters for selection, however, is the noise relative to the difference in candidate quality. We model a comparison of two candidates under Gaussian evaluation noise, calibrating the noise and expected-score gaps from Luna runs (Appendix G.1). At GPQA's estimated typical gap, even 6 measurements per example give only a 65% probability of choosing the better candidate (Figure 5c). A fixed measurement budget therefore becomes less effective as useful improvements become smaller, even if the optimizer continues to discover them.

Methods spend little evaluation budget on confirmation. More than 96% of each method's evaluations on the search split go to new candidates (Appendix Table 37). Our previous analysis instead motivates reserving additional evaluations for close comparisons among promising candidates, including the current best. However, more effort spent resolving existing close comparisons also leaves less budget for discovering new candidates.

More precise search scores do not ensure transfer in isolation. We freeze the starting and returned agents from 12 optimization runs and execute each six times on every original search and test example. We compare the returned agent with its starting agent under these fresh evaluations. While GPQA retains test gains, CharXiv's average test gain is approximately zero (Appendix Figure 17). Repeated measurements improve precision on the search examples, but generalization still requires a separate check on unseen examples (Appendix G.2).

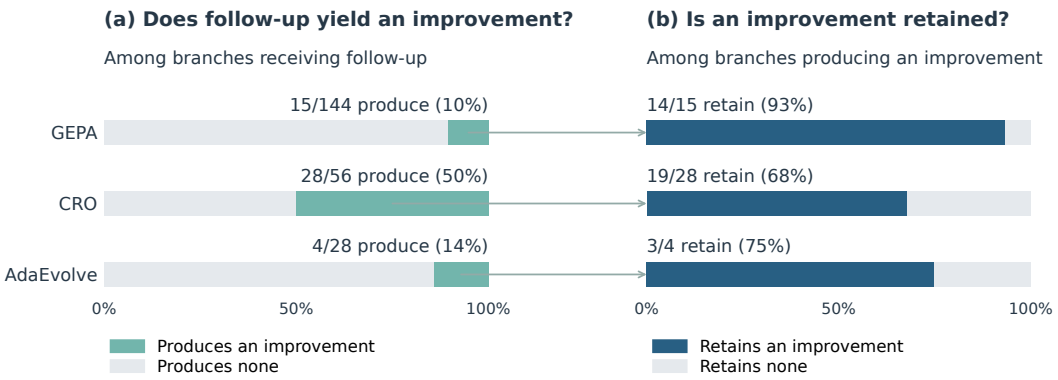

Figure 6: **Further improvements are uncommon, but GEPA retains most when found.** Within twenty subsequent edits: (a) fraction of followed-up branches producing another measured improvement; (b) fraction of those branches retaining at least one. Arrows connect the groups; labels show branch counts. Improvements are relative to each edit's parent (Appendix H.1).

6.3 Selection Stage

Failure to compound gains. Optimizers often continue editing a harness that was improved once, but this work rarely yields another improvement. We trace each candidate that was an improvement over its parent and all later versions derived from it (its *branch*) to see whether further edits produce improvements and whether those improvements are kept. Figure 6 separates these events, revealing different bottlenecks across methods, which we study below. Appendix H gives the full analysis and also examines issues related to *premature commitment* and *coarse credit assignment*.

Parent-relative acceptance can help preserve local improvements. GEPA retains further measured improvement on 14 of the 15 branches that produce at least one. Its design offers a plausible explanation: GEPA screens a child against the particular parent it edits on the same minibatch and admits the child when that local score improves, without requiring it to beat the run's current best. Its per-example Pareto population can then keep candidates that are strongest on particular items available for further search. This design preserves locally promising directions.

An incumbent threshold can discard local gains. CRO produces another measured improvement on 28 of 70 branches, but retains it on only 19. Its rule requires a candidate to at least match the current incumbent. A child can therefore improve the older parent it edits and still be rejected.

6.4 Modeling Optimization Progress

Can proposal effects and evaluation noise together explain the progress observed during optimization? We combine these measured quantities in a simple model of the optimization loop and test its predictions on held-out runs. Once fitted using measured edits of two parent harnesses from one run, the model uses only another run's initial evaluation to estimate its subsequent search gains. Using the score notation from §3, let $S = S_{\mathcal{D}_{\text{search}}}(h)$ and $S' = S_{\mathcal{D}_{\text{search}}}(h')$ be the expected search scores of a parent and its child:

$$\underbrace{\delta \sim q(\cdot \mid S), \quad S' = S + \delta}_{\text{propose}} \qquad \underbrace{\hat{S}' = S' + \varepsilon}_{\text{evaluate}} \qquad \underbrace{\text{retain if } \hat{S}' > \hat{S}_{\text{best}}}_{\text{select}}. \tag{1}$$

The model draws an edit effect δ from a distribution q whose mean depends on the parent's score, adds evaluation noise ε, and keeps the child if its observed score beats the highest score recorded so far, $\hat{S}_{\text{best}}$. The accepted child then becomes the next parent and sets the new score to beat. We estimate noise from repeated evaluations of the seed harness h_0 and fit the edit distribution separately for each base model, task, optimizer, and method (full grouping rules in Appendix J.5). We compare model and run at the same number of evaluations. The two fitting parents can come from later stages of the run used for fitting, so these measurements are not necessarily available at its start.

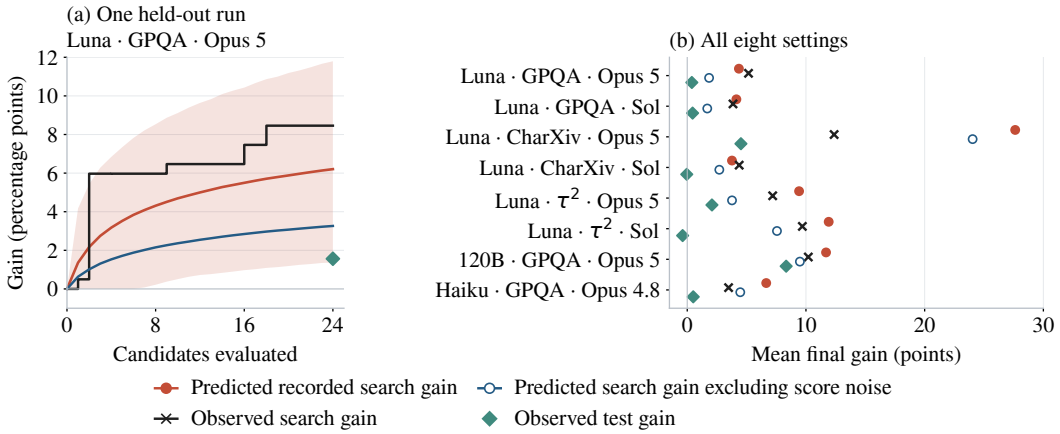

Figure 7: **The model captures search progress more closely than test gains.** (a) One held-out run, selected to illustrate good agreement; shading shows the model's 80% prediction interval. (b) Mean final gains across all eight Meta-Harness settings (38 runs). Predicted gains excluding score noise still concern the search examples. Full results: Appendix Table 43.

Across 38 held-out Meta-Harness runs, the model's estimates of search gain differ from the observed gains by 2.51 percentage points on average (mean absolute error); 29/38 (76.3%) fall within its 80% prediction intervals. Figure 7 shows one run's progress and mean final gains in all eight settings, including failures. The model substantially overpredicts Luna/Opus/CharXiv. Agreement is also weaker for test gains: even the model's expected search gains with score noise removed need not transfer to unseen examples. The results support edit effects and noisy selection as a partial explanation of observed search progress. Appendix J provides the full model and analysis, including fitting details, convergence simulations, measurement costs, all-setting results, and failure analyses.

7 Implications for Future Meta-Optimization Methods

The failures identified above suggest specific changes to how optimizers propose edits, measure progress, and build on discoveries. The following directions connect those findings to testable design choices. Appendix I develops targeted proposals for all 9 failure modes.

Broaden proposals and ground them in failures. Repeated use of proposer prior-driven interventions, together with low improvement yield, suggests that just proposing more edits is insufficient. Proposer training and search procedures could encourage distinct, task-specific hypotheses about observed failures. Before editing, an optimizer could inspect relevant traces, compare explanations, and gather targeted evidence that distinguishes them. Evaluation should then ask whether the resulting interventions repair those failures and improve independently measured performance under a matched budget.

Make progress independently verifiable. Favorable score noise can inflate both apparent gains and the threshold that later candidates must beat. Optimizers could reserve part of their budget to reevaluate promising candidates and incumbents, improving the accuracy of the score used as a bar for selection over time. However, such checks only establish accuracy on the search set, and separate examples are still needed to assess generalization.

Preserve and develop useful changes. The meta-optimizer's tendency to follow-up on early promising candidates without producing another improvement suggests that continued editing does not by itself produce cumulative gains. Instead, optimizers could independently track supported interventions, test their removal or combination, and use those results to choose what to preserve and which candidates to develop. Comparing these choices under matched budgets would help distinguish poor parent selection from ineffective follow-up proposals.

AI USE STATEMENT

We used AI coding and writing assistants to help develop software, analyze experimental records, and draft and revise the manuscript. Language models also serve explicit roles in the study: as optimizer and worker agents, as benchmark judges and simulated users, and as annotators of interventions in code edits. We describe these roles, model configurations, annotation procedures, and validation checks in the paper and appendices. Model-generated annotations are measurements with possible errors, rather than human-verified ground truth. The authors are responsible for the final manuscript, analyses, and claims.

ETHICS STATEMENT

Our experiments study agent optimization on benchmark tasks, including simulated customer-service interactions. Improving these systems could reduce development costs, but could also make unreliable or harmful behavior easier to scale. In particular, a rising search score can conceal evaluation noise or behavior that does not transfer to new inputs. Our results support independent evaluation before deployment; neither benchmark gains nor predicted search gains establish that an agent is safe for real-world use. Released traces and annotations are intended to make optimization decisions auditable and to support research on more reliable evaluation and selection.

REPRODUCIBILITY STATEMENT

We will publicly release all code, data, and artifacts produced for and accompanying this paper: the MOT implementation and method integrations; starting agents, prompts, configurations, environment specifications, and run manifests; recorded candidate programs, optimization traces, per-item evaluation outcomes, and cost records; the MOT-Trace annotations, annotation prompts, and validation artifacts; and the analysis, model-fitting, and plotting scripts with the inputs needed to regenerate every reported table and figure. Some recorded traces contain test-set information, directly or indirectly; we will redact this information from the publicly released artifacts. The release will identify the exact run subsets, exclusions, task splits, random seeds, and recorded software and model versions used in each analysis, and include instructions for reproducing the analyses from the archived outputs and for launching new runs. For upstream benchmarks and models, we will provide source references and loading instructions, with redistribution governed by their original terms. Recomputing results from the released records is distinct from rerunning hosted models: provider changes, unavailable model versions, and sampling nondeterminism can prevent exact replication of a new run.

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. *arXiv preprint arXiv:2507.19457*, 2025.

Jean-Yves Audibert and Sébastien Bubeck. Best arm identification in multi-armed bandits. In *Proceedings of the 23rd Conference on Learning Theory*, pp. 41–53, 2010.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

Robert E. Bechhofer. A single-sample multiple decision procedure for ranking means of normal populations with known variances. *The Annals of Mathematical Statistics*, 25(1):16–39, 1954.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. AdaEvolve: Adaptive LLM driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026. doi: 10.48550/arXiv.2602.20133. URL https://arxiv.org/abs/2602.20133.

Mengzhuo Chen, Junjie Wang, Zhe Liu, Yawen Wang, Haiming Zheng, and Qing Wang. From failed trajectories to reliable llm agents: Diagnosing and repairing harness flaws. *arXiv preprint arXiv:2606.06324*, 2026.

Yizhe Chi, Deyao Hong, Dapeng Jiang, Tianwei Luo, Kaisen Yang, Boshi Zhang, Zhe Cao, Xiaoyan Fan, Bingxiang He, Han Hao, Weiyang Jin, Dianqiao Lei, Qingle Liu, Houde Qian, Bowen Wang, Situ Wang, Youjie Zheng, Yifan Zhou, Calvin Xiao, Eren Cai, and Qinhuai Na. Frontier-eng: Benchmarking self-evolving agents on real-world engineering tasks with generative optimization. *arXiv preprint arXiv:2604.12290*, 2026.

Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ, second edition, 1988.

Herbert A. David and Haikady N. Nagaraja. *Order Statistics*. Wiley, Hoboken, NJ, third edition, 2003.

Shiyi Du, Jiayuan Liu, Weihua Du, Yue Huang, Jiayi Li, Yingtao Luo, Xiangliang Zhang, Vincent Conitzer, and Carl Kingsford. Why search when you can transfer? Amortized agentic workflow design from structural priors. *arXiv preprint arXiv:2604.25012*, 2026.

Olive Jean Dunn. Multiple comparisons among means. *Journal of the American Statistical Association*, 56(293):52–64, 1961.

Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. The reusable holdout: Preserving validity in adaptive data analysis. *Science*, 349(6248):636–638, 2015.

Lucheng Fu, Ye Yu, Yiyang Wang, Yiqiao Jin, Haibo Jin, B. Aditya Prakash, and Haohan Wang. TextReg: Mitigating prompt distributional overfitting via regularized text-space optimization. *arXiv preprint arXiv:2605.21318*, 2026.

Xingze Gao, Chuanrui Hu, Hongda Chen, Pengfei Yao, Zhao Wang, Yi Bai, Zhengwei Wu, Yunyun Han, Xiaofeng Cong, Jie Gui, Yafeng Deng, and Teng Li. EvoAgentBench: Benchmarking agent self-evolution via ability transfer. *arXiv preprint arXiv:2607.05202*, 2026.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. In *International Conference on Learning Representations*, 2024. arXiv:2310.01798.

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). In *Advances in Neural Information Processing Systems*, 2025.

Yaochu Jin and Jürgen Branke. Evolutionary optimization in uncertain environments—a survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317, 2005.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement. *arXiv preprint arXiv:2607.15524*, 2026a.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026b.

Minhua Lin, Juncheng Wu, Zijun Wang, Zhan Shi, Yisi Sang, Bing He, Zewen Liu, Tianxin Wei, Zongyu Wu, Zhiwei Zhang, Dakuo Wang, Xiang Zhang, Benoit Dumoulin, Cihang Xie, Yuyin Zhou, Suhang Wang, and Hanqing Lu. Harness updating is not harness benefit: Disentangling evolution capabilities in self-evolving LLM agents. *arXiv preprint arXiv:2605.30621*, 2026.

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, 2026. doi: 10.1145/3786335.3813221.

Lovish Madaan, Aaditya K. Singh, Rylan Schaeffer, Andrew Poulton, Sanmi Koyejo, Pontus Stenetorp, Sharan Narang, and Dieuwke Hupkes. Quantifying variance in evaluation benchmarks. *arXiv preprint arXiv:2406.10229*, 2024.

Oded Maron and Andrew W. Moore. Hoeffding races: Accelerating model selection search for classification and function approximation. In *Advances in Neural Information Processing Systems 6*, pp. 59–66, 1993.

Evan Miller. Adding error bars to evals: A statistical approach to language model evaluations. *arXiv preprint arXiv:2411.00640*, 2024.

Jinming Nian, Zhiyuan Peng, Hongwei Shang, Dae Hoon Park, and Yi Fang. Submodular evaluation subset selection in automatic prompt optimization. *arXiv preprint arXiv:2601.03493*, 2026.

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. *arXiv preprint arXiv:2506.13131*, 2025.

Qwen Team. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet classifiers generalize to ImageNet? In *International Conference on Machine Learning*, 2019.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A graduate-level Google-proof Q&A benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Chengshuai Shi, Kun Yang, Zihan Chen, Jundong Li, Jing Yang, and Cong Shen. Efficient prompt optimization through the lens of best arm identification. *arXiv preprint arXiv:2402.09723*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

James E. Smith and Robert L. Winkler. The optimizer's curse: Skepticism and postdecision surprise in decision analysis. *Management Science*, 52(3):311–322, 2006.

Yuda Song, Hanlin Zhang, Carson Eisenach, Sham Kakade, Dean Foster, and Udaya Ghai. Mind the gap: Examining the self-improvement capabilities of large language models. In *International Conference on Learning Representations*, 2025.

Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, Andrei Lupu, Roberta Raileanu, Kelvin Niu, Tatiana Shavrina, Jean-Christophe Gagnon-Audet, Michael Shvartsman, Shagun Sodhani, Alexander H. Miller, Abhishek Charnalia, Derek Dunfield, Carole-Jean Wu, Pontus Stenetorp, Nicola Cancedda, Jakob Nicolaus Foerster, and Yoram Bachrach. AI research agents for machine learning: Search, exploration, and generalization in MLE-bench. *arXiv preprint arXiv:2507.02554*, 2025.

Varun Ursekar, Apaar Shanker, Veronica Chatrath, Yuan Xue, and Samuel Marc Denton. VeRO: A harness for agents to optimize agents. *arXiv preprint arXiv:2602.22480*, 2026a.

Varun Ursekar, Apaar Shanker, Yash Maurya, Shehab Yasser, Vijay S. Kalmath, Veronica Chatrath, and Yuan Xue. HarnessOpt-Bench: Evaluating LLMs at harness optimization. *arXiv preprint arXiv:2608.06301*, 2026b.

Yike Wang, Huaisheng Zhu, Zhengyu Hu, Yige Yuan, Zhengyu Chen, Shakti Senthil, Hannaneh Hajishirzi, Yulia Tsvetkov, Pradeep Dasigi, and Teng Xiao. Rethinking the evaluation of harness evolution for agents. *arXiv preprint arXiv:2607.12227*, 2026.

Zirui Wang, Mengzhou Xia, Luxi He, Howard Chen, Yitao Liu, Richard Zhu, Kaiqu Liang, Xindi Wu, Haotian Liu, Sadhika Malladi, Alexis Chevalier, Sanjeev Arora, and Danqi Chen. CharXiv: Charting gaps in realistic chart understanding in multimodal LLMs. *arXiv preprint arXiv:2406.18521*, 2024.

Peng Xia, Rujun Han, Zifeng Wang, Yanfei Chen, Yufan Zhang, Yoonho Lee, Chengsong Huang, Han Yu, Zhongying CuiZhu, Yifei Ming, Huaxiu Yao, Burak Gokturk, Tomas Pfister, and Chen-Yu Lee. RRSI: Regularized recursive self-improvement of agent harnesses. *arXiv preprint arXiv:2609.24972*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W. Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 2369–2380, 2018.

Simon Yu, Derek Chong, Ananjan Nandi, Dilara Soylu, Jiuding Sun, Christopher D. Manning, and Weiyan Shi. Shepherd: Enabling programmable meta-agents via reversible agentic execution traces. *arXiv preprint arXiv:2605.10913*, 2026a.

Ye Yu, Xiaopeng Yuan, Haibo Jin, Heming Liu, Yaoning Yu, and Haohan Wang. Do self-evolving agents forget? capability degradation and preservation in lifelong llm agent adaptation. *arXiv preprint arXiv:2605.09315*, 2026b.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. TextGrad: Automatic "differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024. arXiv:2310.02304.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=z5uVAKwmjf.

Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R. Tomz, Christopher D. Manning, and Weiyan Shi. Verbalized sampling: How to mitigate mode collapse and unlock LLM diversity. *arXiv preprint arXiv:2510.01171*, 2025c.

Xing Zhang, Yanwei Cui, Guanghui Wang, Ziyuan Li, Wei Qiu, Bing Zhu, and Peiyang He. Library drift: Diagnosing and fixing a silent failure mode in self-evolving llm skill libraries. *arXiv preprint arXiv:2605.19576*, 2026a.

Xing Zhang, Guanghui Wang, Yanwei Cui, Wei Qiu, Ziyuan Li, Bing Zhu, and Peiyang He. Prompt optimization is a coin flip: Diagnosing when it helps in compound AI systems. In *ICML Workshop on Combining Theory and Benchmarks (CTB)*, 2026b.

APPENDIX CONTENTS

A Future Work and Limitations . . . **15**

B Configuration . . . **16**
B.1 Framework Coverage Beyond the Main Grid . . . 20

C The Main Result by Factor . . . **21**
C.1 Additional Practical Baselines . . . 22

D Labelling Interventions . . . **25**
D.1 Expanded Annotation Coverage and Comparability . . . 26
D.2 Independent Human Annotation Audit . . . 28

E Summary of Failure Modes . . . **30**

F Proposal: Further Results . . . **30**
F.1 Concentration and Measured Outcomes . . . 30
F.2 No-Feedback Proposals from Each Optimizer . . . 32
F.3 Do Shown Failures Support the Proposed Intervention? . . . 33
F.4 Instructions and Edit Scope . . . 35
F.5 Feedback and Proposal Choice . . . 39
F.6 Retrying Rejected Ideas . . . 41
F.7 Rapidly Diminishing Returns . . . 42
F.8 Full Inventory of Frequent Interventions . . . 43

G Evaluation: Further Results . . . **46**
G.1 Repetition Budget and Selection Reliability . . . 46
G.2 Independent Reevaluation of Returned Agents . . . 48
G.3 Measured Effects and Selection Noise . . . 51
G.4 Allocation of Evaluation Effort . . . 52
G.5 Overtrusting Dev Scores . . . 53
G.6 Evaluation Overhang . . . 55

H Selection: Further Results . . . **55**
H.1 Ancestry and Continuation Windows . . . 56
H.2 Premature Commitment and Parent Choice . . . 57
H.3 Coarse Credit Assignment . . . 58

I Research Directions by Failure Mode . . . **59**
I.1 Proposal: Generate and Test Better Ideas . . . 59
I.2 Evaluation: Separate Apparent Progress from Confirmed Gains . . . 59
I.3 Selection: Develop and Preserve Useful Discoveries . . . 60

J Validation of Search Dynamics: Full Model and Analysis . . . **60**
J.1 From Testbed Observations to a Model . . . 60
J.2 How Proposal Effects and Noise Shape Progress . . . 61
J.3 Validation on Runs Excluded from Fitting . . . 61
J.4 Noise, simulation, and illustrative regimes . . . 63
J.5 Calibration Cohorts and Measurement Budget . . . 63
J.6 Separate Qwen Two-Parent Check . . . 65
J.7 Evaluation Noise in the Main and Side Grids . . . 66

K Selection-Noise Derivations and Testbed Measurements . . . **67**
K.1 Setup . . . 67
K.2 Noise in one validation score . . . 67
K.3 Comparing two candidates . . . 68
K.4 Choosing among T candidates . . . 68
K.5 How many reruns a decision needs . . . 69
K.6 Headroom and progress . . . 69
K.7 Fitting the gain model to GEPA's proposals . . . 71

L Supplementary Trace Summaries . . . **73**
L.1 The Life Table . . . 73
L.2 Kinds of Intervention . . . 74
L.3 Runs as Sequences of Edits . . . 75

GUIDE TO THE APPENDIX

The appendix provides protocols, complete comparisons, and checks behind the main claims. Each entry below points to a specific reviewer question; the supporting failure modes follow the same proposal–evaluation–selection order as Section 6.

What is held constant, and what may each method change?
Table 2: design counts and the distinct cohorts used by each analysis. Appendix B: method interfaces and settings, task splits, starting agents, cost limits, jail coverage, and held-out-data access audits. Appendix C: gain definitions, exclusions, and results by factor.

Which methods and benchmarks does MOT support beyond this study?
Appendix B.1: current method and benchmark integrations, the earlier 17-configuration/10-family inventory, and the distinction between framework coverage and the main experimental grid.

What do the intervention labels measure, and how reliable are they?
Appendix D: annotation inputs, identity matching, derived operations, and checks. Appendix D.2: returned independent human reviews, agreement, unresolved cases, and the whole-edit display limitation. Appendices L.1–L.3: intervention retention, kinds, and run trajectories. The nine-pattern taxonomy is Table 19.

Do proposals reflect a narrow prior or the failures shown?
Appendices F.1–F.8: concentration, optimizer-matched no-feedback controls, the completed common/non-common evidence check, and every common intervention's description and run frequency. Appendices F.4 and F.5: instructions, edit scope, inspected examples, and the controlled feedback probe. Appendix F.4.1: the instruction/scope pilot. Retrying rejected ideas and rapidly diminishing returns are examined in Appendices F.6 and F.7.

How much can scores and selection decisions be trusted?
Appendix G: selection noise and evaluation allocation. Appendix G.2: independent reevaluation of starting and returned programs, complete results, and serving-route sensitivity. Appendices G.5 and G.6: overtrusting dev scores and evaluation overhang. Appendix K.5: same-item remeasurements in the smaller-model testbed.

Are improvements developed, and which changes matter?
Appendix H.1: ancestry, the distinction between producing and retaining further improvements, window sensitivity, and missing-parent coverage. Appendices H.2 and H.3: parent choice and intervention-removal experiments, with their attribution limits.

How could future optimizers address the nine failure modes?
Appendix I: a numbered roadmap linking each failure mode to concrete model or method changes and the evidence needed to assess them.

How does the statistical model validate the account of search dynamics?
Appendix J: the full model and validation analysis. Appendix J.5: reserved runs, parent selection, measurement counts, and all-setting results (Table 43). Appendix J.6: the separate Qwen check. Appendix J.4: simulation equations and illustrative regimes. Appendix K: derivations and earlier testbed measurements.

Reading the cohorts. The 288-run design, the expanded 212-run annotation dataset (202 Luna and ten weaker-worker runs), and each analysis's eligible cohort are different quantities. Captions and protocols give their denominators and exclusions. Appendix D.1 explains which measures pool across annotation batches; identity counts and fixed validation experiments retain their stated cohorts. The intervention audit and the historical evidence-judge check validate different measurements; neither validates subsequent judges. Appendices D.2 and F.3 distinguish these checks.

A FUTURE WORK AND LIMITATIONS

Why do capable agents struggle to reliably optimize other agents? Our analysis locates recurring failures in how an opinionated proposer forms, updates, and acts on beliefs about effective strategies. Using MOT to study available runs from a 216-run main-grid design across four methods and three benchmarks, we identify nine failure modes across proposal, evaluation, and selection. Proposals repeatedly return to familiar interventions, noisy evaluations provide unreliable evidence for

revising them, and selection can reinforce early choices without producing further gains. Tracking interventions makes these failures visible beyond a run's final score.

These findings motivate methods that help proposers revise their prior beliefs using execution evidence, distinguish reliable gains from favorable evaluations, and build on useful interventions. Progress should be measured by whether successive edits produce improvements that persist on unseen examples. MOT makes these questions accessible to systematic study by recording what changes, how it is evaluated, and what the optimizer retains. We will release the testbed and all accompanying code, data, traces, and annotations to support developing and testing such methods.

Limitations. The human audit supports agreement on several annotation dimensions, but code assignments still require adjudication and whole-edit coverage requires an operation-aware review (Appendix D.2). Test scores are noisy, and current analyses cover provisional subsets of the planned grids. Methods use released settings and often stop before exhausting their budgets. The prediction model's selection assumptions and uncertainty omissions limit its scope (§6.4); it does not predict transfer to unseen items.

B CONFIGURATION

Which records support each result? The experimental design, labelled dataset, and analysis denominators answer different questions. A run can contribute to an intervention analysis while lacking the comparable start/end evaluations required by a gain analysis. Counts below overlap and must not be added together. The main performance results use the 216-run main grid; annotation-specific analyses use the 212 runs with completed annotations.

Table 2: **Cohorts used in the paper.** Design counts describe planned experimental cells; all other rows describe the records used in the indicated analysis. Each cited protocol supplies the inclusion rules and exclusions.

Evidence	Count	Scope and source
Experimental design	216 main; 72 side	Luna main grid and weaker-worker GPQA side grid (Table 4); these are not completion counts.
Main performance	216 Luna runs	Four methods across three benchmarks, three starting agents, two optimizer models, and three seeds (Figure 3a; Table 10).
Worker comparison	42 / 9 / 6 runs	GPQA subsets for Luna / Haiku / gpt-oss-120B; settings are unmatched (Figure 3b).
MOT-Trace	212 runs	202 Luna and ten weaker-worker runs; 20,392 labelled edits and 11,350 intervention records (Table 13).
Proposal concentration	185 Luna runs	At least nine labelled edits per run: 65 CharXiv, 59 GPQA, 61 τ^2-bench (Figure 4; Table 14).
Continuation	330 candidates	219 GEPA, 70 CRO, 41 AdaEvolve candidates with eligible ancestry and complete twenty-edit windows (Figure 6; Appendix H.1).
Human annotation audit	180 cases	120 interventions, 40 identity pairs, 20 whole edits; two reviewers each, first annotation batch only (Appendix D.2).
Independent reevaluation	12 runs	Frozen starting/returned harness pairs on GPQA and CharXiv, six fresh repetitions per item (Appendix G.2).
Model validation	38 target runs	Meta-Harness runs in eight settings, excluding fitting runs (Figure 7; Table 43).

Table 3: **How the four methods implement the loop.** Each row compares one design choice (settings in Table 7). A screen is a preliminary evaluation before full scoring. Meta-Harness keeps candidates on the accuracy–context-length frontier: none is both less accurate and longer than another retained candidate. †On τ^2-bench, CRO edits only the prompt.

	proposer: LLM call		proposer: coding agent	
	GEPA	AdaEvolve	Meta-Harness	CRO
PROPOSE				
edits	one prompt	program diff	any file	program stages†
sees	minibatch traces	scores only	full history	training traces
EVALUATE				
screen	random batch	none	none	picked items
scored on	search set	training set	training, search	search set
SELECT				
keeps if	beats parent	outranks a similar program	on the frontier	$\geq$ incumbent
parent	best on individual items	adaptive subpopulation choice	agent's choice	agent's choice
returns	best search	best training	best search	best search

Selection objective and score precision. Let $\mathcal{A}$ contain the candidates scored for final selection. With comparable per-attempt variances, averaging K independent attempts on N fixed items reduces score standard deviation as $1/\sqrt{KN}$; it does not remove differences between item sets. Writing D_{sel} for the method's selection set (training for AdaEvolve, search for the others), the returned candidate is

$$h^\star \in \arg\max_{h \in \mathcal{A}} \hat{S}_{D_{\text{sel}}}(h), \tag{2}$$

where $\mathcal{A}$ contains the candidates scored for final selection. Our quantity of interest is the test gain $S_{D_{\text{test}}}(h^\star) - S_{D_{\text{test}}}(h_0)$. A high selection score need not imply a large test gain: selecting among noisy readings favors overestimates, and adapting to the selection items can yield improvements that do not transfer to unseen items (overfitting). Section 5.1 defines how we measure search and test gains, including when the highest search reading belongs to a different candidate from the one returned.

This appendix distinguishes the experimental design, the performance-analysis subset, and the expanded annotation dataset (Table 2). Configuration details for those snapshots are read from run records or the code at each recorded commit; where they disagree, we report the run records.

Design. The main grid crosses four methods, two optimizer models, three task families, three seed agents per family, and three random seeds: $4 \times 2 \times 3 \times 3 \times 3 = 216$ runs with GPT-5.6-Luna as the base model (Table 4). A separate 72-run GPQA-Diamond side grid varies the base model, giving 288 runs in the experimental design. The performance figures report the eligible recorded subset described in Appendix C; the design count does not imply full coverage in those comparisons.

Earlier completion snapshot. In the earlier 186-run main-grid snapshot, 108 GPT-5.6-Sol runs and 78 Claude Opus 5 runs had completed or been sealed; 21 AdaEvolve, 6 Meta-Harness, and 3 CRO runs from the Opus arm were not yet included. Of those 186 runs, 103 ended when the method stopped, 75 at the wall-clock limit, 6 when the process was killed, and 2 at the budget limit. Each was sealed on its current incumbent for test evaluation. These counts describe that snapshot, not completion of the full design. This historical completion snapshot is not the denominator for the current gain or annotation analyses; their counts are listed separately in Table 2.

Weaker base models. The side grid crosses two base models (Claude Haiku 4.5 and gpt-oss-120B), four methods, three seed agents (minimal, medium, strong), and three random seeds, $2 \times 4 \times 3 \times 3 = 72$ runs, on GPQA-Diamond with Claude Opus 5 fixed as optimizer. The matched reference is the 36-run GPQA-Diamond/Claude Opus 5 subset of the main grid, with GPT-5.6-Luna as base model and the same method, seed-agent, and random-seed settings. Thus the base-model

comparison contains 108 runs in the design, 36 per base model. It uses the same task splits and three repetitions per search and test item. The current worker-model panel contains 42 Luna, nine Haiku, and six gpt-oss-120B runs. Luna pools optimizer models, whereas the weaker-worker groups use Opus; this panel is descriptive and does not implement the matched 108-run design.

Table 4: **Factors and fixed settings of the main grid.**

Factor	Levels
Task family	GPQA-Diamond; τ^2-bench (airline and retail); CharXiv (reasoning questions)
Seed agent	minimal, medium, strong per family (Table 5)
Method	GEPA, AdaEvolve, Meta-Harness, CRO (Table 7)
Optimizer model	Claude Opus 5; GPT-5.6-Sol
Base model	GPT-5.6-Luna; the separate GPQA-Diamond side grid uses Claude Haiku 4.5 and gpt-oss-120B
Random seed	0, 1, 2
Setting	**Value**
Repetitions per item	3 on the search and test sets; 1 on the training set
Budget	\$150 per run, optimizer model and agent metered together
Wall-clock limit	72 hours per run (24 hours for the CRO runs with Claude Opus 5)
Rollout timeout	600 seconds
Sampling	provider defaults for the base and optimizer models; no temperature or token limit is set

Tasks. Each family is split once into training, search, and test sets, and every seed agent and run of a family uses the same split. *GPQA-Diamond* (198 questions; 65/67/66): four-option multiple choice with the options shuffled per question under a fixed seed, scored by exact match on the chosen letter. *τ^2-bench* (all 164 airline and retail tasks; 54/55/55, with airline 16/17/17 and retail 38/38/38): tool-using customer-service dialogues against a simulated user, scored by the benchmark's reward over the final database state, required actions, and natural-language assertions, with at most 50 steps. *CharXiv* (200 reasoning questions over scientific charts, 25 from each of 8 subjects; 68/66/66): the split balances subject, answer type, and question source; answers are graded 0/1 by a language-model judge with CharXiv's per-type rubric. The CharXiv judge and the τ^2-bench user simulator and assertion judge are GPT-5.4-mini for every run, rather than the models used in the original benchmarks, so scores are comparable within our grid but not with published numbers.

Table 5: **Seed agents.** Each is a small program around the base model; the optimizer may edit any part of it that the method exposes.

	minimal	medium	strong
GPQA-Diamond	one call from a system prompt to a letter	self-refine: draft, critique, revise (three calls)	three reasoning branches with different stances, then one call that evaluates and votes
τ^2-bench	the benchmark's tool-calling agent loop under an editable policy prompt	the same loop with a reflect-first policy	the same loop with a plan-then-act policy
CharXiv	one vision call	a draft, then a vision call that verifies and revises it	a vision call that writes a Python program over the values it reads, which is then executed

Models. The base model GPT-5.6-Luna, and the optimizer models when they are called directly, are served through one OpenAI-compatible proxy; Claude Haiku 4.5 and gpt-oss-120B are served through Amazon Bedrock. When the method's proposer is a coding agent (Meta-Harness and CRO), it is Claude Code for Claude Opus 5 and the Codex command-line agent for GPT-5.6-Sol, each at its default settings. Model calls are retried up to four times on transport errors.

Evaluation. An evaluation on the search or test set runs each item three times and records the mean score and every repetition; training evaluations run once. After a run, the test set is scored on the returned harness $h^\star$; the test gain in the main text is measured against the seed agent's pooled test level for that task and seed agent, computed from every test reading of the seed, so that it does not depend on one noisy reading. Rollouts run concurrently (2 to 64 at a time, depending on the run); concurrency does not change what is scored.

Table 6: **Pooled test baselines for the nine original starting agents.** Scores are proportions on the held-out test set. The spread and standard error are across independent measurements; all methods use the same reference for a given task and starting tier.

Task	Starting tier	Pooled score	SD	SE	Measurements
CharXiv	Minimal	0.7174	0.0217	0.0050	19
CharXiv	Medium	0.7082	0.0327	0.0087	14
CharXiv	Strong	0.7216	0.0187	0.0047	16
GPQA-Diamond	Minimal	0.8430	0.0205	0.0045	21
GPQA-Diamond	Medium	0.8631	0.0117	0.0028	18
GPQA-Diamond	Strong	0.8770	0.0157	0.0035	20
τ^2-bench retail	Minimal	0.5466	0.0403	0.0101	16
τ^2-bench retail	Medium	0.5537	0.0404	0.0108	14
τ^2-bench retail	Strong	0.5155	0.0518	0.0126	17

Table 7: **Method settings in the grid.** Each method runs from its released code; values not listed are the method's defaults.

Method	Settings
GEPA	Released engine, unmodified. Optimizes the seed agent's prompt files, one component at a time in round-robin order. Reflection minibatch of 8 training items (5 on τ^2-bench); candidates scored on the full search set and chosen from the per-item Pareto front; merge off. Stops on its budget or the wall-clock limit.
AdaEvolve	Released engine. Two starting islands, up to five; population 20; islands chosen by an upper-confidence bound with decay 0.9; migration of 5 programs every 15 iterations. 300, 250, and 200 iterations on GPQA-Diamond, CharXiv, and τ^2-bench. Fitness and selection use the training-set mean; the search set is scored for our records only. Every scored child enters its island's archive of 20 programs; once the archive is full, it replaces the most similar program outside the protected top 20% if its elite score, 0.7 times its fitness-rank percentile plus 0.3 times its code-novelty percentile, is higher. The incumbent is the program with the best training score.
Meta-Harness	Reference implementation. 18, 40, and 24 iterations on GPQA-Diamond, CharXiv, and τ^2-bench, two candidates per iteration; a coding-agent session of up to 120 turns per iteration. The proposer reads training-set traces of every earlier candidate; the frontier ranks candidates by search-set accuracy and context length, which is our one change to the reference loop (it originally ranks on the training items the proposer reads).
CRO	Released snapshot. 12 iterations on GPQA-Diamond and 20 on CharXiv; on τ^2-bench 12 for the Claude Opus 5 runs and 24 for the GPT-5.6-Sol runs. Batches of 4 counterfactual variants (2 on τ^2-bench), screened on 8 to 16 training items with no net drop allowed; a candidate is promoted if its search score is at least the incumbent's. Coding-agent sessions of up to 200 turns.

Access to test data. In the earlier snapshot, we scanned every logged tool call of the Meta-Harness and CRO proposers, the only ones that use tools (99 runs, about 1.2 million calls). None opens a split file, a task file, or a benchmark data file. We also searched every tool output for the identifiers of the run's search and test items: none appears, apart from identifiers cut off at an output limit and ones the proposer typed itself. The proposers could nonetheless read beyond their workspace: 36 runs searched the host's file system, 39 read the framework's source code, and 10 read files of an unrelated optimization project on a shared volume. None of these reads reached our tasks' test data. The isolation coverage and its limits are described below.

Isolation. In the earlier completion snapshot, 66 of the 186 runs ran their method inside the filesystem jail of §4; the other 120 ran without it because the jail's dependency was unavailable on their host, and are labelled as such in the release. In both groups the evaluator withholds test results and workspace provisioning omits search and test instances. The method-process jail restricts filesystem writes outside the workspace but permits network access; this write restriction alone does not establish that task files elsewhere on the host are unreadable. The file-access audit above is separate evidence about observed behavior, not a guarantee against every route to held-out data.

B.1 Framework Coverage Beyond the Main Grid

The experimental grid is a subset of MOT's implemented integrations. Tables 8 and 9 distinguish the current framework from an earlier version of the infrastructure. "Current" denotes an implemented method adapter or benchmark runner and dataset interface; it does not imply that every method–benchmark pairing has been evaluated. The main grid contains GEPA, AdaEvolve, Meta-Harness, and CRO on GPQA-Diamond, τ^2-bench, and CharXiv.

Historical counting. The earlier roster contains 17 exported optimizer entry points, including a no-op baseline, separate coding-agent backends, and a wrapper that selects between those backends. We therefore call them *optimizer configurations*, rather than 17 distinct algorithms. Its 10 benchmark families group AgentDojo's base, attack, and defense configurations together, exclude the mixed-QA carrier, and count each benchmark once regardless of seed-agent variants. These counts refer to the same repository snapshot (`e3b62e36`); they are neither cumulative counts across versions nor a claim that the current release retains every historical integration. The historical roster records integration availability, not a completed 17×10 evaluation. Similarly named adapters can differ across versions.

Table 8: **Method integration inventory.** The upper panel lists current adapters; "Grid" marks the four methods in the main comparison. The lower panel enumerates all 17 entries in the earlier exported roster. Historical entries are not additional methods evaluated in this paper.

Implementation	Search procedure / role	Grid
Current framework		
GEPA	Reflective prompt search	Yes
AdaEvolve	Adaptive evolutionary program search	Yes
Meta-Harness	Coding-agent search over recorded history	Yes
CRO	Counterfactual program/prompt optimization	Yes
AFlow	Search over code-defined workflows	—
HGM adapter	A candidate's own code proposes its successor	—
Coding-agent baseline	Autonomous editing and evaluation in a session	—
Prompt hill climbing	Failure-guided prompt revision	—
Earlier roster: 17 exported configurations		
Hermes	Self-evolution of skills and memory	
Agentic CBO (Codex)	Coding-agent proposals with persistent history	
Agentic CBO (Claude Code)	Alternate proposer backend for agentic CBO	
ADAS	Search over agent programs	
CAPO	Cost-aware prompt optimization with racing	
Claude Code	Coding-agent source-revision baseline	
Codex	Alternate coding-agent source-revision baseline	
TRIPLE	Bandit-based prompt selection	
GEPA	Reflective prompt search	
Meta-Harness wrapper	Model-based dispatch to Claude Code or Codex	
Autonomous coding agent	Agent controls its edit–evaluate loop	
AdaEvolve	Adaptive evolutionary program search	
EvoX	Evolution of programs and search strategies	
Null	No-op baseline; retains the starting agent	
MaAS	Search over multi-agent architectures	
DGM	Archive-based self-improving coding agents	
SICA	Self-improving coding-agent lineage	

Table 9: **Benchmark integration inventory.** "Current" lists active benchmark integrations in the inspected framework; "Earlier" refers to the 10-family historical roster. "Grid" marks the paper's main experimental tasks. Benchmark families are not multiplied by domains, dataset splits, or starting agents.

Benchmark family	Task domain	Current	Earlier	Grid
GPQA-Diamond	Science question answering	Yes	Yes	Yes
τ^2-bench	Interactive customer-service tools	Yes	Yes	Yes
CharXiv	Visual chart reasoning	Yes	—	Yes
Terminal-Bench-2	Terminal coding and system tasks	Yes	Yes	—
AIME	Competition mathematics	Yes	—	—
LongCoT	Long-context reasoning	—	Yes	—
SWE-bench Verified	Repository issue resolution	—	Yes	—
OfficeQA-Pro-open	Document retrieval and answering	—	Yes	—
AgentDojo	Tool-use security; base/attack/defense	—	Yes	—
SOL-ExecBench	GPU-kernel optimization	—	Yes	—
ALE-Bench	Heuristic programming optimization	—	Yes	—
Math-Discovery	Mathematical construction search	—	Yes	—

Terminal-agent support. Terminal-Bench-2 uses a Harbor runner with an editable worker agent, benchmark-owned task environments and verifiers, and Docker or Modal execution backends. MOT records verifier rewards, terminal trajectories, and worker usage alongside optimization cost. This integration enables research on terminal-agent optimization under the same recording and evaluation interfaces. Terminal-Bench-2 is outside the main grid; the present failure-mode findings have not been established on terminal tasks.

Inventory boundaries. The historical benchmark roster follows manifests marked active with a ready harness in the pinned snapshot. In the current tree, BIG-Bench Hard data remain available but its manifest marks it retired, so it is excluded from active coverage. The inventories also exclude vendored source without an active adapter and other retired experimental integrations.

C The Main Result by Factor

How the gains are measured. The search gain of a run is its best full search-set reading minus its first reading of the seed agent, and the test gain is the returned agent's test score minus the seed agent's pooled test level for that task and seed agent (§5.1; Appendix B). Nineteen Meta-Harness runs were resumed after an interruption, and their record names the candidate at the point of resumption as the run's start; we take the search baseline from the seed agent's first reading in the original segment and retain the pooled original-seed test baseline, and leave out two τ^2-bench runs whose first segment did not start from the seed. The spend per run in Figure 3 is the metered cost of the optimizer model and the agent together, averaged over the method's runs; test evaluation is not charged.

CRO runs left out. CRO converts the seed agent into its own program format before the search. In 10 of its 51 runs the converted seed's first search reading lies far below the other methods' reading of the same seed agent (for instance 66 to 71 points against 89 on GPQA-Diamond with the strong seed, and under 7 points against 62 to 67 on τ^2-bench in five runs), so the search gain of these runs mostly recovers what the conversion broke. We keep the 41 runs whose converted seed reads within 15 points of the reference; for these the median difference is 3.6 points.

Table 10: **The main result by factor**: search gain and test gain in points, mean ± standard deviation over runs. n is the number of analyzed runs in the recorded subset, after the exclusions above; it is not the full-grid completion count (216 main-grid runs and 72 side-grid runs in the design). Runs resumed after an interruption are measured from the seed agent, not from the point of resumption; two τ^2-bench runs that never started from the seed are excluded. The CRO row covers its 41 runs whose converted seed reads within 15 points of the other methods' reading of the same seed agent; the task, optimizer, starting-agent, and pooled rows use the 96 non-CRO Luna runs. The worker-model rows are separate GPQA subsets, as specified below.

	n	search	test		n	search	test
method				*optimizer model*			
GEPA	54	3.1 ± 3.0	0.6 ± 2.9	Claude Opus 5	51	4.8 ± 3.8	1.8 ± 3.8
Meta-Harness	30	6.3 ± 4.1	1.8 ± 4.0	GPT-5.6-Sol	45	3.2 ± 3.2	0.1 ± 2.2
AdaEvolve	12	3.0 ± 2.6	0.7 ± 2.2	*seed agent*			
CRO	41	5.8 ± 3.4	-0.6 ± 3.9	minimal	40	4.5 ± 3.4	0.9 ± 2.5
task family				medium	28	4.1 ± 3.6	0.7 ± 2.5
GPQA-Diamond	42	3.5 ± 2.6	0.3 ± 1.9	strong	28	3.3 ± 3.8	1.5 ± 4.6
CharXiv	28	3.7 ± 3.5	0.9 ± 2.2	*base model, GPQA-Diamond*			
τ^2-bench	26	5.3 ± 4.7	2.3 ± 5.1	GPT-5.6-Luna	42	3.5 ± 2.6	0.3 ± 1.9
pooled subset, excluding CRO				Claude Haiku 4.5	9	4.6 ± 1.3	2.2 ± 3.2
pooled	96	4.1 ± 3.6	1.0 ± 3.3	gpt-oss-120B	6	6.4 ± 4.7	3.6 ± 5.4

C.1 Additional Practical Baselines

Shared evaluation. Both baselines use GPT-5.6-Luna as worker and the same sealed GPQA-Diamond and CharXiv test splits as the main grid: 66 items per task, with three repetitions per item. GPQA uses the grid's normalized exact-match scorer; CharXiv uses its GPT-5.4-mini judge. All reported endpoints have exactly three outcomes for each of the same 66 items. The three designed seeds are reevaluated on the same direct OpenAI endpoint in the same evaluation period. This supports comparisons with those seeds; scores from the earlier main-grid serving period are not treated as a matched comparison.

Table 11: **Practical baselines on the same held-out items.** Test accuracy (%), using Luna as worker, 66 items per task and three repetitions per item. Seeds are reevaluated on the same endpoint in the same evaluation period. Each free-coding condition has one optimization run; its blank start differs from the main grid. Sol spends \$31/\$37 and Opus \$190/\$166 on GPQA/CharXiv against a nominal \$150 limit. Paired uncertainty and protocol details: Appendix C.1.

Agent	GPQA	CharXiv
Minimal seed	82.3	71.7
Medium seed	86.9	72.2
Strong seed	86.4	73.2
Luna in Codex CLI (no optimization)	82.3	70.7
Free coding agent: Codex + Sol	87.4	76.3
Free coding agent: Claude Code + Opus 5	90.9	77.3

Luna in Codex CLI, without optimization. Codex CLI 0.155.1 runs a separate headless session for each test item, with Luna, the minimal seed's task instruction, and the question (plus the chart for CharXiv). Each rollout has a fresh workspace and configuration directory, with web search disabled and shell network blocked. Answers use the minimal seed's extraction and the grid's grader. Failed rollout processes are retried at the same item/repetition index. Codex makes no tool calls on GPQA and uses tools in about 6% of CharXiv sessions. This baseline therefore tests this particular ready-made harness, not the benefit of tool use in general.

A free coding agent constructs the harness. Codex with GPT-5.6-Sol and Claude Code with Opus 5 each receive an empty agent stub, the input/output and worker-model interfaces, MOT's evaluation commands, and a nominal \$150 budget. They may inspect training feedback and request

full or subsampled search scores; test evaluation is unavailable during optimization. No edit strategy or candidate-selection loop is prescribed. Sessions resume from the same workspace and notes until the budget is exhausted or three consecutive sessions request no evaluation. Each optimizer–task combination has one run. Worker evaluations use concurrency eight, a 600-second timeout, and three repetitions per search/test item, under a 72-hour overall ceiling; Sol additionally has a two-hour limit per session.

The reported runs use filesystem isolation and block web-search tools. An earlier Sol attempt accessed test traces and is excluded entirely; only its isolated reruns are reported. The Opus process retains network access for API calls, but an audit found no network commands among 1,052 shell calls, and scans of all four returned agents found no training/search/test item text. These checks do not establish exhaustive absence of leakage. Opus can inspect framework source while Sol cannot, so differences between the arms combine optimizer and scaffold differences.

Cost and uncertainty. Sol spends $31.46 on GPQA and $37.14 on CharXiv before reaching the idle stopping rule. Opus spends $190.15 and $165.78: in-flight session charges arrive after the budget check, exceeding the nominal cap by 27% and 11%. Two failed Sol/CharXiv sessions have unmetered optimizer charges, so its recorded cost is incomplete. Optimization costs use the main grid's accounting rates; the no-optimization baseline's separately recorded inference costs use different rates and are not compared here.

Table 12: **Paired test differences for the practical baselines.** Percentage-point difference $\pm$ one standard error, computed from the 66 paired item means after averaging each item's three repetitions. These errors describe item-level variation and do not measure variability across optimization runs.

Comparison	GPQA	CharXiv
Luna in Codex minus minimal seed	0.00 ± 2.69	-1.01 ± 2.38
Free coding: Sol minus strong seed	1.01 ± 2.03	3.03 ± 2.67
Free coding: Opus minus strong seed	4.55 ± 3.87	4.04 ± 3.91

The results establish feasible alternatives, not a ranking under matched starts and realized costs. In particular, the blank-start runs cannot be pooled with the main grid's gains over designed seeds. Differences between search and test scores also mix execution noise and generalization; they do not by themselves identify winner's curse.

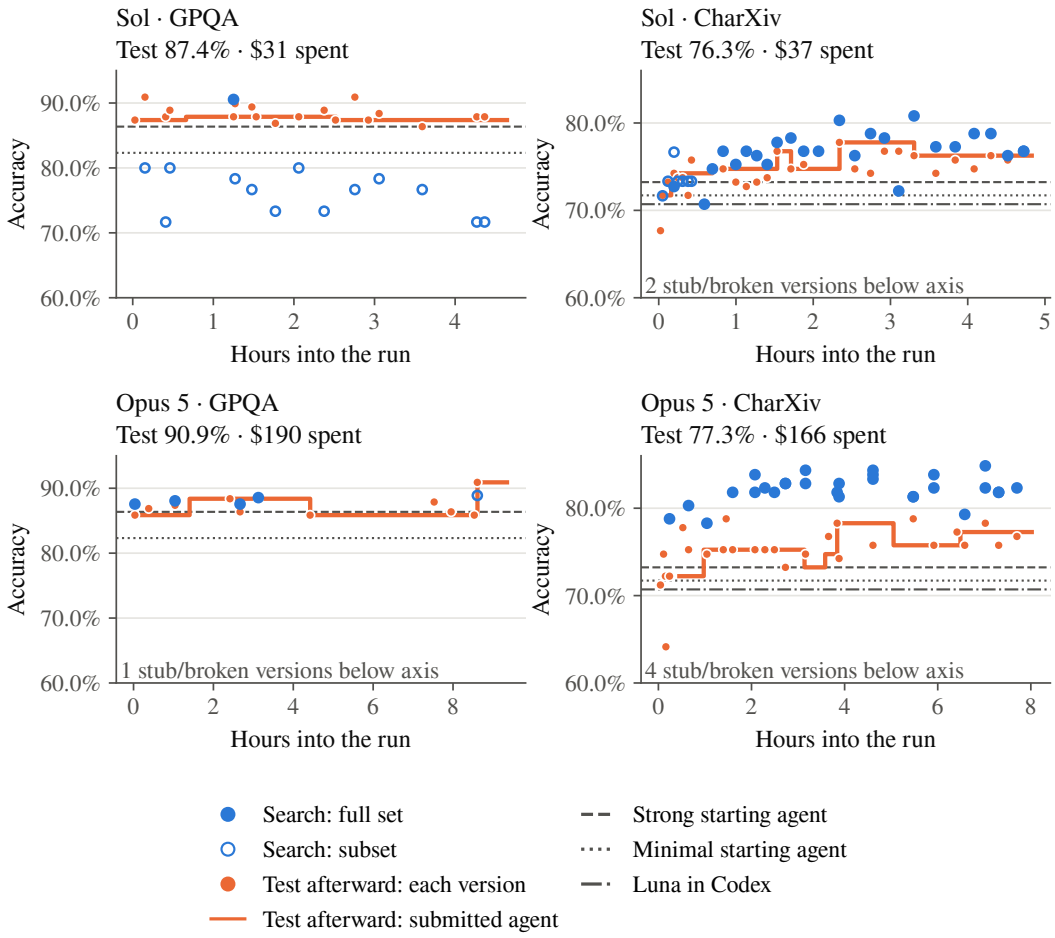

Figure 8: **Exploratory free-coding trajectories.** Blue points are scores available during search: filled points use the full search set, open points a subsample. Orange points are post-hoc test evaluations of archived programs, and the orange line follows the submitted program; test scores were not available to the optimizer. Horizontal lines show reevaluated reference agents. Each panel represents one run. The minimal seed and Luna in Codex tie on GPQA and share the dotted reference line. The vertical axes omit low-scoring stub/broken versions, whose counts are indicated; scales differ across tasks. Spend is recorded optimization cost, with the overshoots and missing charges described above.

D LABELLING INTERVENTIONS

The intervention analyses in Section 6 use optional post-run annotations of MOT's saved records. The score and cost comparisons in Section 5.2 use the run records directly and do not require these labels. This appendix describes the labelling pipeline used in this study, planned for integration as an optional MOT module, and its checks. Intervention identities and additional flags are model-assigned; operations, survival, and measured edit effects are derived from the record.

Runs covered. The expanded annotation dataset contains 212 runs, 20,392 labelled edits, and 11,350 intervention records: the original 137 runs plus 75 later runs. The Luna subset contains 202 runs (54 GEPA, 53 Meta-Harness, 46 AdaEvolve, and 49 CRO); ten additional labelled runs use weaker workers. The intervention analyses and gain analyses use overlapping but different cohorts. In particular, the late export's provisional gains do not replace the canonical gain tables. Coverage, batch comparability, and run-quality qualifications are reported in Appendix D.1.

The record. Every edit of every run is stored with its diff against the candidate it was built on, the prompt or agent session that produced it, the loop's verdict, and every evaluation of the base and the child with per-item scores. A run is replayed in order; the interventions it holds are tracked candidate by candidate, so that for any edit we know which interventions its base already contains. Meta-Harness does not declare parents. MOT records the most similar earlier candidate by content, and we replace it with the base the proposer actually copied, read from the copy command in its session. In the first batch, this recovers the base for 54% of Meta-Harness edits, and for those the content-based guess is wrong 60% of the time; analyses of parent choice use only recovered bases and mark the other edits *unknown*.

Pass 1: intervention identity. A language model (Claude Sonnet 4.6, temperature 0 for the first batch; Sonnet 5, temperature 1 for the late batch) reads the edits of a run in order. For each edit it receives the diff, the run's *registry* of interventions so far (id, where in the agent it lives, what change it makes, one-line summary), which of those the base contains, and a mechanical cut of the diff into units (a prompt bullet, a code statement, a constant) matched against the base. It returns, for each intervention the edit changes, either an existing registry id or a new entry. The rubric fixes what counts as one intervention: one place in the agent together with one functional change made there; the same place and change in different words or code is the same intervention; changing a parameter of it is a change to it, not a new one; several rules serving one place and change are one intervention. The model is trusted only for *which* intervention an edit touches. The operation (introduce, develop, remove, reintroduce) is then derived from membership: first appearance in the run, present in the base, absent from the base but known to the run. In the first batch, edits whose answer was cut off at the output limit (276 of 12,841) were re-asked with a larger limit and the full registry. That first pass covers 137 runs, 12,841 edits, and 9,117 interventions, at a median of 39 interventions per run.

Pass 2: identity across runs. Runs on the same task name the same intervention differently. For each task family, one call built a codebook of intervention-level entries from the registry descriptions (65 for CharXiv, 90 for GPQA-Diamond, 99 for τ^2-bench), merging rewordings, re-implementations, and parameter variants into one entry; every intervention was then assigned one code. Both batches use these same task codebooks and Sonnet 4.6 for code assignment. Codes are used only where interventions are compared across runs (§6.1, §6.3). The kinds of Figure 27 are a grouping of the codebook entries into eight kinds, assigned once per entry.

Pass 3: item-specific. Sonnet 4.6 labels each intervention in both batches *item-specific* or *general* from its registry entry. The rubric: an intervention is item-specific only if it would not fire on a fresh task item unless that item were nearly identical to one the optimizer saw (a specific answer, value, entity, product, figure, or the exact wording of one question); a rule scoped to a recurring kind of item (a question type, a domain, a chart type, a policy area) is general. Within the first batch, a first version of this rubric, which let rules for a question type count as item-specific, flagged 2.5 times as many interventions; the two versions agree on 82% of interventions and disagree in one direction only (1,592 flagged by the first and not the second, 15 the reverse).

Checks. The following automated checks concern the first batch and test consistency and related signals; the independent human audit follows below. (i) Pass 1 against the mechanical cut: of 12,840 operations, 8 contradict the units (an intervention marked introduced whose text the cut had seen before, or the reverse). (ii) Pass 3 against an independent signal: of interventions whose edit was labelled, in a separate pass, as transcribing content from a task item, 84% (GEPA) and 91% (Meta-Harness) are flagged item-specific; none of CRO's, which copies nothing, are. (iii) Pass 2 against the analyses that use it: the overlap results of §6.3 are relative comparisons between relations (replicate, same method, different method) against a null that preserves each code's popularity, but can still depend on how the codebook splits or merges interventions.

Cost. For the first batch, Pass 1 cost $479 and Passes 2 and 3 under $10 together. The planned release includes intervention identity, item-specific flags, task codes, and derived quantities linked to the run records.

D.1 Expanded Annotation Coverage and Comparability

The second annotation release adds 75 runs to the first 137: 7,551 edits and 2,233 intervention records, for totals of 212 runs, 20,392 edits, and 11,350 records. The late batch contains 65 Luna runs and ten weaker-worker runs. The Luna subset therefore contains 202 runs and 19,478 edits; the main code-level concentration analysis retains 185 runs with at least nine edits. This minimum-length rule is unchanged from the earlier figures.

Table 13: **Annotation coverage by batch.** Intervention counts describe the output of each identity labeller; their sum is a dataset size, not a common-granularity measure of diversity.

Batch	Runs	Luna runs	Labelled edits	Intervention records
First: Sonnet 4.6 identities	137	137	12,841	9,117
Late: Sonnet 5 identities	75	65	7,551	2,233
Combined	212	202	20,392	11,350

Which quantities can be pooled? Identity extraction changed from Sonnet 4.6 at temperature 0 to Sonnet 5 at temperature 1. On a matched 66-run comparison, Sonnet 5 produced about 0.42 times as many intervention identities per labelled edit (per-run ratio interquartile range 0.31–0.60). Consequently, identity counts, first appearances, attempts per identity, operation shares, and life-table measures are not pooled. The original identity-dependent figures retain the first batch; the retry and item-specific summaries below show both batches separately. Task-code assignment and item-specific flags use Sonnet 4.6 with the same codebooks and rubric in both batches. We pool code-presence and edit-level measured-effect analyses under this shared coding procedure. This is an analysis choice, not a validation of cross-batch equivalence: omissions and identity granularity can still affect detected code coverage, and the first-batch human audit does not bound those effects in the late batch. Item-specific percentages retain identity-weighted denominators and are therefore shown by batch rather than treated as a homogeneous pooled rate.

Cohorts and run quality. The 202 Luna runs comprise 54 GEPA, 53 Meta-Harness, 46 Ada-Evolve, and 49 CRO runs. The late release includes 44 newly completed canonical runs, 12 Databricks-worker runs, one additional AdaEvolve run, eight τ^2-bench AdaEvolve runs stopped at a 16-hour ceiling, and ten weaker-worker runs. Rate limits restricted child production in the Databricks-optimizer AdaEvolve runs to 2–48% of iterations; empty iterations contain no diff and are absent from the labelled-edit tables. Three late CRO runs carry worker-failure flags (11–24% of rollouts). We retain these annotations in descriptive coverage and report a sensitivity excluding late runs with recorded flags or Databricks-worker provenance. These cohorts cannot be interpreted as budget-matched or route-matched method comparisons. Seven additional weaker-worker runs remain unsealed and are excluded; "expanded" does not mean every planned run is fully labelled.

Table 14: **Concentration under alternative inclusion rules.** Entries give run count, number of frequent codes, and median percentage of edits touching at least one frequent code. Frequent sets are recomputed within each task and cohort. The first row reproduces the original figure; the second is the updated main analysis.

Cohort	CharXiv	GPQA	τ^2-bench
First batch, at least nine edits	38 / 20 / 98.9%	45 / 7 / 80.0%	40 / 13 / 91.3%
Expanded, at least nine edits	65 / 14 / 84.0%	59 / 9 / 80.0%	61 / 8 / 71.1%
Expanded, all run lengths	70 / 13 / 84.9%	63 / 7 / 74.6%	69 / 6 / 60.0%
Expanded, flags/routes excluded	53 / 13 / 84.0%	51 / 9 / 80.5%	53 / 9 / 73.7%

Including shorter runs changes both the frequency threshold and the median edit coverage; the main figure preserves the original minimum-nine-edit criterion. The change is substantive: concentration is lower than in the first batch, especially on CharXiv and τ^2-bench. Both the updated figure and the inclusion sensitivity retain majority edit coverage by a small set of codes.

Continuation sensitivity. Excluding the same flagged/Databricks-worker runs leaves the following follow-up, produced-improvement, and retained-improvement counts: GEPA: 144, 15, 14 out of 219; CRO: 49, 24, 15 out of 59; AdaEvolve: 15, 0, 0 out of 20. These use the same conservative ancestry and twenty-edit window rules as the main figure. Four resumed CRO runs and one AdaEvolve run reset their recorded step counters; ordering by normalized edit position preserves their chronological sequence.

Table 15: **Retry yield reported separately by identity labeller.** Measured-improving edits / measured edits, followed by percentage. All Luna run lengths are included here. First attempts and retries concern identities not yet retained; either group may also modify previously retained identities. No late GEPA runs are available.

Batch	Method	First attempts	Repeat attempts
First	GEPA	88/2579 (3.4%)	25/772 (3.2%)
First	Meta-Harness	48/320 (15.0%)	23/206 (11.2%)
First	AdaEvolve	4/575 (0.7%)	4/851 (0.5%)
First	CRO	62/473 (13.1%)	7/102 (6.9%)
Late	Meta-Harness	3/62 (4.8%)	4/97 (4.1%)
Late	AdaEvolve	17/963 (1.8%)	22/2468 (0.9%)
Late	CRO	66/292 (22.6%)	35/236 (14.8%)

Scope of existing checks. The independent human annotation audit and its code-correction sensitivity concern the first batch. They do not validate the late identity extractor. The evidence-justification comparison, controlled feedback probe, instruction/scope pilot, ablation experiment, paired promotion-test study, evaluation-allocation audit, and endpoint validation retain their original sampled cases and frequency definitions where relevant. The no-feedback suggestion lists are reused without new calls, but their coverage is recomputed against the expanded task–optimizer cohorts. The prediction fits and canonical gain tables are unchanged: the late label export's own-seed provisional gains are not interchangeable with the paper's pooled-seed test gains.

Weaker-worker coverage. The ten sealed weaker-worker runs contain 914 labelled edits. They remain outside the Luna concentration and continuation cohorts. Table 16 reports their edit coverage using the Luna frequent sets, without fitting new sets to these small groups. This is descriptive coverage, not a matched worker-model comparison.

Table 16: **Sealed weaker-worker annotations.** Median fraction of edits touching the frequent set defined from the eligible Luna cohort on that task. Small cells and unsealed runs limit comparisons.

Worker	Task	Method	Runs	Edit coverage (%)
Haiku 4.5	CharXiv	AdaEvolve	3	75.3
Haiku 4.5	CharXiv	GEPA	1	100.0
Haiku 4.5	GPQA	AdaEvolve	3	50.8
Haiku 4.5	GPQA	GEPA	2	95.1
gpt-oss-120B	GPQA	AdaEvolve	1	29.7

D.2 Independent Human Annotation Audit

Sample and procedure. The audit samples the first, Sonnet 4.6 annotation batch; it does not validate the late Sonnet 5 identities. Ten PhD students working on AI agents, all outside the author team, independently reviewed 180 cases, with two different reviewers per case and 36 assignments per reviewer. All 360 reviews were returned. The sample comprises 120 intervention descriptions, 40 within-run identity pairs, and 20 whole-edit inventories, covering 79 runs overall and 68 in the primary intervention sample. The 120 interventions were sampled within task $\times$ method $\times$ machine-specificity strata with approximately equal allocation; population summaries use weights N_h/n_h. The unequal weights alone give an effective sample size of about 53. The identity sample contains 20 annotated later touches and 20 lexically similar negative pairs; it is a diagnostic challenge set, not a population-error sample. Whole edits were sampled separately by task and method.

Reviewers received source diffs and task codebooks, with model codes and specificity flags withheld. For whole edits, each reviewer first wrote an independent inventory, then compared it with the displayed extraction. Returns contain per-case judgments, evidence notes, confidence, and the independent inventories; the coordinator confirmed independent completion and the pre-reveal sequence. Original judgments are preserved. This audit concerns intervention annotations, not the separate evidence-justification judge of Appendix F.3.

Table 17: **Independent reviewers agree on most annotation dimensions.** Raw agreement and nominal Krippendorff's α on distinct double-reviewed cases. Uncertainty remains a response category. These measure reliability, not agreement with the model. Whole-edit judgments concern the displayed inventories, subject to the limitation below.

Dimension	Matching pairs	Agreement (%)	α
Description support	115/120	95.8	.825
Atomicity	118/120	98.3	.935
Item-specificity	116/120	96.7	.920
Task-code assignment	103/120	85.8	task-specific
Within-run identity	39/40	97.5	.940
Whole-edit omissions	20/20	100.0	1.000
Whole-edit unsupported entries	20/20	100.0	1.000
Whole-edit grouping	16/20	80.0	.629

Codes have different meanings across tasks, so we report code-assignment α separately: .860 on CharXiv (45 cases), .857 on GPQA (53), and .810 on τ^2-bench (22). High reviewer agreement does not establish that the original model annotations are correct.

Reference and unresolved cases. Our primary comparison requires matching, non-uncertain labels and both reviewers marking the response clear. The confidence flag applies to the entire response, so uncertainty about one dimension can leave other agreed dimensions unresolved. At least one reviewer reports uncertain confidence for 64 of the 120 interventions. Under the original disagreement-or-uncertainty rule, 66 intervention cases, 13 identity pairs, and seven whole edits require adjudication; none has yet been adjudicated. Unresolved cases are neither counted as correct nor silently discarded from population claims.

Table 18: **Agreement with the model among clear human consensuses.** Raw matches are accompanied by sampling-weighted agreement for interventions. The resolved-weight column shows how much of the primary population weight is represented. Identity uses an unweighted challenge set. Description support and atomicity test the properties claimed by the extracted descriptions.

Dimension	Matches / resolved	Agreement (%)	Resolved weight (%)
Fully supported description	51/55	90.9	56.1
Single intervention	52/55	98.4	54.6
Item-specificity	55/56	99.98	56.1
Task-code assignment	41/56	82.6	56.1
Within-run identity	24/27	88.9	27/40 cases

The five raw agreement fractions are $51/55$, $52/55$, $55/56$, $41/56$, and $24/27$. Their arithmetic mean is 89.5%, but this mixes annotation dimensions and two sampling designs, and uses only clear consensuses. We therefore report the dimensions separately rather than present this mean as overall annotation accuracy. The intervention percentages in Table 18 use the sampling weights; for example, raw task-code agreement is $41/56 = 73.2\%$, while its weighted value is 82.6%. Whole-edit ratings are excluded because of the display limitation below.

These conditional estimates should not be generalized to all cases. Allowing unresolved labels to be either correct or incorrect gives weighted all-sample agreement bounds of 51.0–94.9% for full description support, 53.8–99.1% for atomicity, 56.1–99.99% for specificity, and 46.4–90.2% for codes. These are bounds from unresolved labels, not confidence intervals. As a sensitivity check, accepting matching labels regardless of the overall confidence flag yields weighted agreement of 87.2%, 88.7%, 95.1%, and 72.3%, respectively. Identity agreement becomes 30/39. This looser reference is not our primary validation result.

Sensitivity of proposal concentration. Fifteen of 56 clear code consensuses differ from the model. We replace only these reviewed assignments in memory and recompute the task-level frequent sets and edit shares in the original 123-run concentration cohort. The concentration cohort contains 54 of the reviewed interventions, including 14 changed codes. The common sets remain at 20, seven, and 13 codes for CharXiv, GPQA, and τ^2-bench, and the median edit shares remain 98.88%, 80.00%, and 91.34%. This checks the observed clear-code corrections only; it does not establish robustness to unresolved codes, missing interventions, or identity errors. The archived annotations remain unchanged pending adjudication.

Whole-edit display limitation. Both reviewers flag omissions in 16/20 displayed inventories and unsupported entries in 16/20. However, the packet builder displayed canonical registry descriptions without the current operation or an edit-specific description. Of 56 entries, 24 are developments, 14 reintroductions, four removals, and only 14 introductions; 17/20 inventories contain reused descriptions. A removal can therefore appear under its original "add" description. In cases R041 and R158, for example, the stored operation correctly records removal of a feasibility stage and a gridline-tiling stage, respectively, but the displayed description suggests adding them. The reviewers reasonably flag the mismatch; it cannot be attributed directly to an incorrect operation annotation.

This limitation does not explain away all reported omissions or grouping problems. In R124, both reviewers identify changes absent from an inventory containing only a newly introduced entry. All 40 independent inventories remain useful for comparison against the current edit and operation. We retain the original ratings and require an operation-aware review of all 20 cases before estimating extraction precision or recall. This display issue is specific to the whole-edit comparison; the primary intervention cases use first-appearance diffs, and identity cases explicitly ask whether a later edit touches a reference intervention.

E Summary of Failure Modes

Table 19: **Failure modes of the meta-optimization loop**, organized by proposal, evaluation, and selection, with relevant design choices and directions to test. These patterns can share causes across operators; the grouping does not assign independent causal responsibility. Retrying rejected ideas is descriptive: rejection can be noisy, and a change may work in a different context. Detailed research directions appear in Appendix I.

	failure mode	definition	design choice	implication
Propose	**Mode collapse**	Search stays concentrated on familiar interventions despite low measured yield.	steering; evidence shown to the proposer	test broader proposal sampling
	Retrying rejected ideas	Previously rejected changes keep returning.	record of outcomes the proposer sees	condition retries on measured outcomes
	Rapidly diminishing returns	Further search produces progressively fewer improvements.	stopping rule; budget allocation	let the budget follow measured progress
Evaluate	**Winner's curse**	Selecting high measured scores favors positive measurement errors.	items, repetitions, and candidates per decision	re-measure before promoting or reporting
	Evaluation overhang	A stored overestimated score can make later improvements harder to recognize.	whether the seed and incumbents are re-measured	refresh stored comparison scores
	Overtrusting dev scores	The optimizer selects changes based on gains on a small evaluation set, but those gains often disappear on new examples.	screen size; acceptance rule	check transfer on unseen examples
Select	**Failure to compound gains**	Follow-up rarely produces another retained, measured improvement.	parent rule; follow-up allocation	confirm and develop promising branches
	Premature commitment	Further effort goes toward candidates before their gains are established.	parents chosen by score	choose parents by estimated true improvement
	Coarse credit assignment	Whole edits are accepted without establishing which component interventions contribute.	unit of credit	test and preserve useful interventions

F Proposal: Further Results

Figure 4a,c uses 65 CharXiv, 59 GPQA, and 61 τ^2-bench runs with at least nine annotated edits. The cohort is the expanded Luna subset (Appendix D.1). Frequent sets are recomputed within task; unmeasured effects are excluded from (c), and its two groups may contain different runs. Improvement-rate intervals bootstrap per-run rates with 20,000 draws (seed 41). Panel (b) uses 15 no-feedback suggestion lists per task and optimizer, with frequent sets defined from that optimizer's own runs (Table 20).

F.1 Concentration and Measured Outcomes

An intervention is *frequent* on a task if at least half of the task's runs propose it; all remaining observed interventions are *less frequent*. Some archived plot labels use "playbook" for this frequent set.

Search concentration and measured outcomes. In Figure 4, the median share of annotated edits involving frequent interventions is 84.0% on CharXiv, 80.0% on GPQA, and 71.1% on τ^2-bench; the frequent sets contain 14, nine, and eight codes, respectively. Among measured edits involving

frequent interventions, the mean per-run improvement rates are 6.3%, 10.2%, and 9.1%; corresponding rates for other edits are 5.0%, 7.2%, and 7.7%. Each rate gives equal weight to runs with measured edits in that group. The groups contain different runs and parents, so these are descriptive comparisons rather than effects of choosing frequent interventions. The late annotations reduce the earlier concentration estimates on CharXiv and τ^2-bench; repeated use remains prevalent, but is less overwhelming than in the first snapshot.

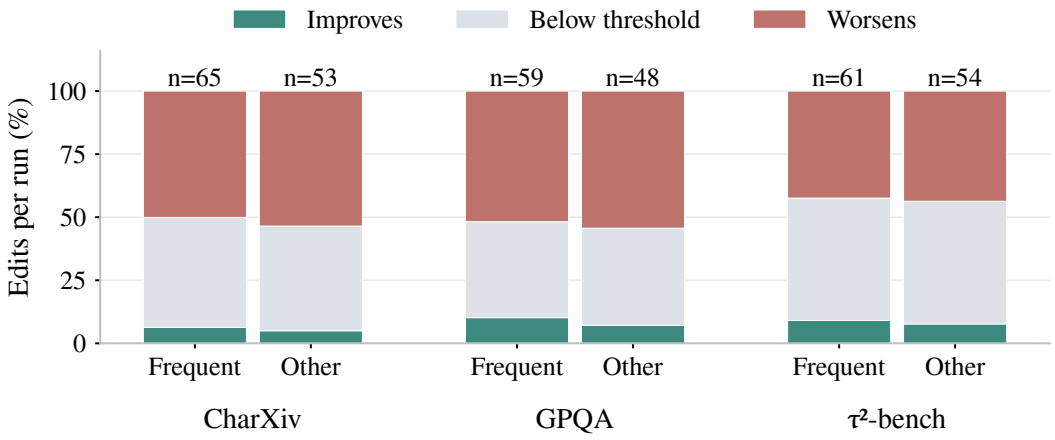

Figure 9: **Measured outcomes of edits involving frequent interventions.** "Frequent" denotes edits using at least one frequent intervention; "Other" denotes the remaining edits. Bars average outcome shares equally across runs; n gives contributing runs. Improvement or worsening requires a score change exceeding one estimated standard error; "Below threshold" does not establish no effect. Groups can contain different runs. Frequent sets and inclusion rules match Figure 4a,c.

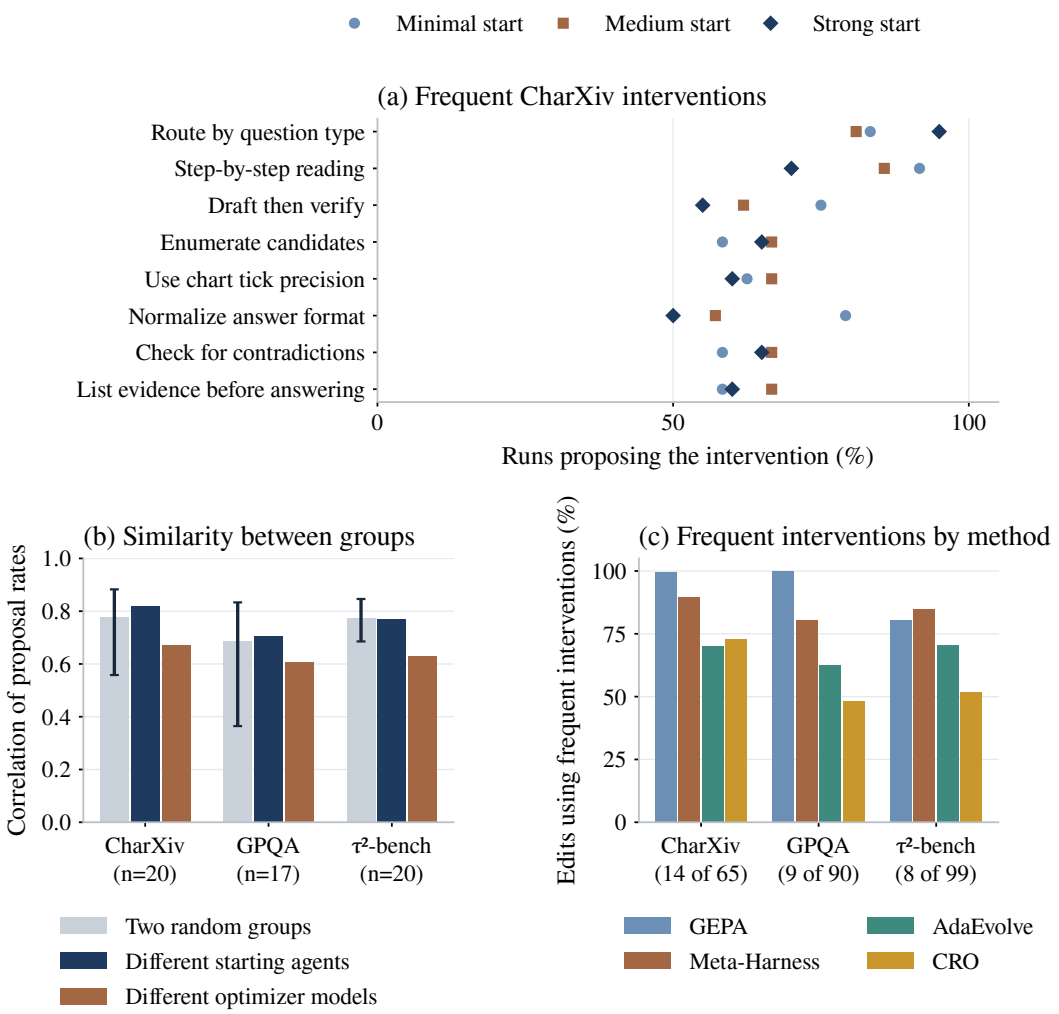

Figure 10: **Intervention prevalence across starting agents and optimizers.** (a) CharXiv's eight most-proposed interventions and the fraction of runs from each starting tier proposing them. (b) Correlation of intervention proposal rates between two random groups, groups with different starting agents, and groups with different optimizer models. Random-group whiskers span the middle 95% of 300 draws. Groups have equal size n within each task; methods are pooled. (c) Median fraction of each method's edits using frequent interventions; labels give the frequent set's size out of all intervention categories on that task. All panels use the expanded Luna cohort with at least nine annotated edits, including AdaEvolve on τ^2-bench.

F.2 No-Feedback Proposals from Each Optimizer

Inputs and outputs. Each optimizer receives the starting agent's source and a short task description, with no task instances, scores, execution traces, or optimization history. We request 8–12 suggested changes in each of five independent samples per starting agent (15 lists per task and optimizer). Suggestions are mapped to the task's intervention codebook, retaining item-specific and unmatched suggestions separately. The comparison below uses observed intervention codes named at least once across those lists; repeated mentions do not increase coverage.

Matching each optimizer to its own runs. We compare the no-feedback suggestions with observed intervention codes from that optimizer's runs on the main worker model, retaining runs with at least nine edits. Within each task–optimizer cohort, frequent interventions occur in at least half of runs; all other observed intervention codes are less frequent. Each model contributes five samples

for each of the three starting agents per task. The outcome is coverage of distinct codes, not the fraction of individual suggestions that are frequent. With frequency groups recomputed on the expanded annotation cohort, both optimizers cover a larger share of frequent codes on all three tasks. The suggestion lists are unchanged; the expanded run cohort changes the frequency groups and their denominators. Opus 5 includes its Databricks-route runs, so these are descriptive model-family comparisons rather than route-matched tests.

Table 20: **No-feedback suggestions compared with each optimizer's own runs.** Number of distinct observed intervention codes named at least once, divided by the number in the corresponding frequency group. Percentages in parentheses. Each task has 15 independent suggestion lists per optimizer. Frequency groups differ between optimizer cohorts and from the pooled groups elsewhere in the paper.

Optimizer	Task	Runs	Frequent	Less frequent
Sol	CharXiv	33	10/11 (91%)	17/54 (31%)
Sol	GPQA	31	6/7 (86%)	21/81 (26%)
Sol	τ^2-bench	26	8/8 (100%)	25/82 (30%)
Opus 5	CharXiv	32	14/18 (78%)	8/46 (17%)
Opus 5	GPQA	28	11/15 (73%)	17/72 (24%)
Opus 5	τ^2-bench	35	10/13 (77%)	27/85 (32%)

The Opus probe produced 552 suggestions in 45 completed samples, including 38 coded as item-specific and seven without a matching codebook entry. Although the prompt requested 8–12 suggestions, samples contained 11–14; all were retained. The generation output limit was raised from 2,000 to 8,000 tokens after truncated responses, retaining earlier complete samples. Intervention coding used Sonnet 4.6; three empty coding responses covering 36 suggestions were repaired with Sonnet 5 using the same rubric and codebook. These repairs concern intervention identity and are separate from the human-validated evidence-justification judgments. Sampling and output settings differ between optimizer models, so the coverage rates are descriptive, not a controlled comparison of model capability. Legacy τ^2 rows with an unspecified worker field are included under the study's main-worker assignment.

F.3 Do Shown Failures Support the Proposed Intervention?

Question and comparison. The no-feedback probe asks whether an optimizer can name an intervention without seeing failures. This separate check asks whether the failures actually shown at a proposal step support that intervention. Frequency groups and sampling are frozen to the first annotation cohort; the expanded labels do not redefine this completed comparison. For each task, we randomly sample up to ten eligible frequent and ten eligible less frequent intervention codes, requiring at least six recorded GEPA introductions. Frequent codes occur in at least half of the pooled task's runs; less frequent codes are their complement, including intermediate-frequency codes. For each code, we sample up to fifteen introduction steps and fifteen control steps where it is absent and not introduced. Controls come from before the code's first appearance in a run or from runs that never propose it. Frequency groups and introduction status are observational comparisons, not randomized treatments.

Blinded judgment. Claude Sonnet 5 (temperature 1) receives the intervention description, the current instruction, and the minibatch items with worker outputs and feedback. It does not see the optimizer's proposed edit or whether this is an introduction step. It assesses whether at least one shown failure could plausibly have been avoided by the intervention. Cases where it judges the intervention already present in the instruction are excluded. All 1,462 requested judgments completed; none is missing or counted as a negative because of a request failure. The retained evidence builder caps text at 30,000 characters and replaces inline image data with placeholders. Missing visual content and unobserved worker reasoning limit what this check can establish.

Table 21: **Supporting evidence at introduction and control steps.** Support judgments divided by eligible cases after excluding interventions judged already present in the instruction. Percentages pool cases; codes and runs can contribute multiple cases. The proportions are descriptive, without treating those cases as independent replicates.

Task	Intervention group	Control steps	Introduction steps
CharXiv	Frequent	32/118 (27%)	74/121 (61%)
	Less frequent	12/130 (9%)	57/130 (44%)
GPQA	Frequent	23/51 (45%)	28/53 (53%)
	Less frequent	11/131 (8%)	38/93 (41%)
τ^2-bench	Frequent	24/104 (23%)	17/64 (27%)
	Less frequent	23/100 (23%)	15/50 (30%)

What the comparison supports. On CharXiv, both groups receive substantially more support at introduction steps. On GPQA the separation is larger for less frequent interventions; frequent interventions often receive support even when not introduced. On τ^2-bench both groups have low support and little separation. The check therefore does not establish that frequent interventions are uniformly less justified than less frequent ones, or that proposers universally ignore feedback. It shows that introduction does not reliably identify a step with supporting failure evidence. A negative judgment need not mean the intervention is useless, and a positive one does not prove that it repairs the actual cause of the worker's error. Eligibility favors interventions with repeated introductions, so these rates do not estimate support across every observed intervention.

Human agreement and historical provenance. Human annotations agreed with the earlier Sonnet 4.6 evidence judge in 98% of 100 sampled cases. That historical check used the same evidence-justification question, with the intervention choice hidden from the judge; it does not validate this Sonnet 5 rerun or the intervention-identity labels. The earlier study compared frequent with rare interventions (fewer than a quarter of runs), omitting intermediate-frequency codes. The completed comparison above uses the full less frequent sampling frame and one judge version throughout. The earlier judgments remain in the analysis archive; the table above is the evidence comparison used in the paper.

F.4 Instructions and Edit Scope

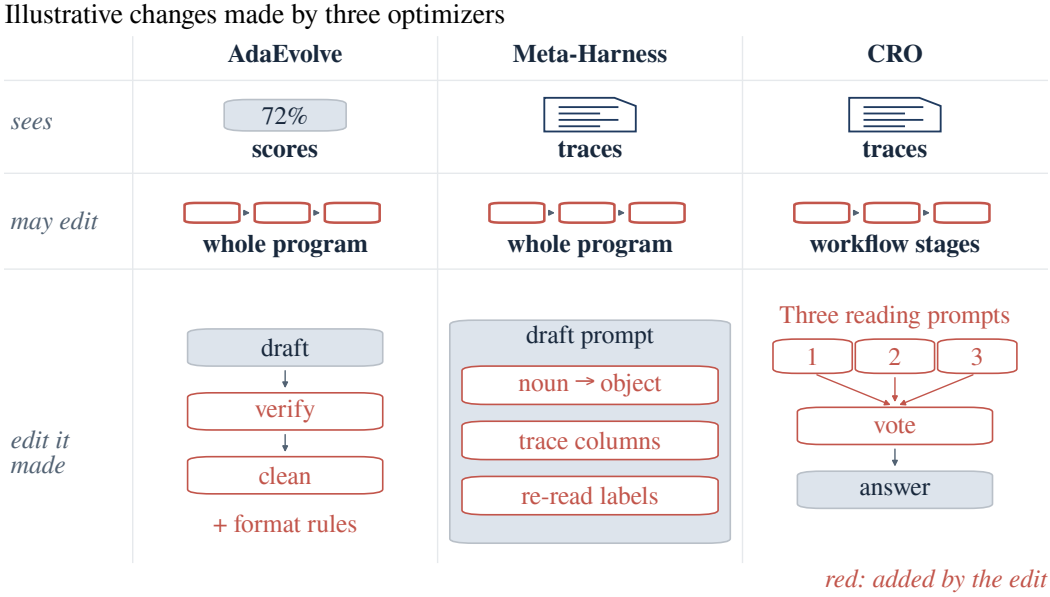

Figure 11: **Concrete examples of changes proposed under different search designs.** Each column shows one kept CharXiv edit that measurably improved the program it changed; red marks additions. Meta-Harness and CRO receive training traces and answers, while AdaEvolve receives scores. These cases illustrate, rather than isolate, differences in feedback and permitted edits. Intervention prevalence is shown separately in Figure 10.

Figure 12 compares how many intervention categories GEPA, CRO, and Meta-Harness explore in their first nine edits. Table 22 gives excerpts from retained proposer prompts. These directions address the optimizer, rather than the worker agent; task profiles and prompt versions can differ across runs.

Table 22: **What each method asks the proposer to do**, quoted from the prompts used in the grid.

Method	What it sees	What it is asked (excerpt)
GEPA	the current prompt; a minibatch of training items with the agent's outputs and feedback	"Identify all niche and domain specific factual information about the task and include it in the instruction."
AdaEvolve	the program and its training score	"Techniques worth exploring: … self-consistency or multi-sample voting, decomposition, verification/refinement passes" (GPQA-Diamond; CharXiv and τ^2-bench have their own lists)
Meta-Harness	every earlier candidate's source and scores, and raw training traces with the answers	"read the traces of low-scoring candidates to see HOW they failed … Do not rely on scores alone."
CRO	training traces and the variants tried so far	"fix failure classes, not literal train examples"; "If local edits plateau, add or split tasks, change control flow"; requires alternative interventions and predicted trace changes

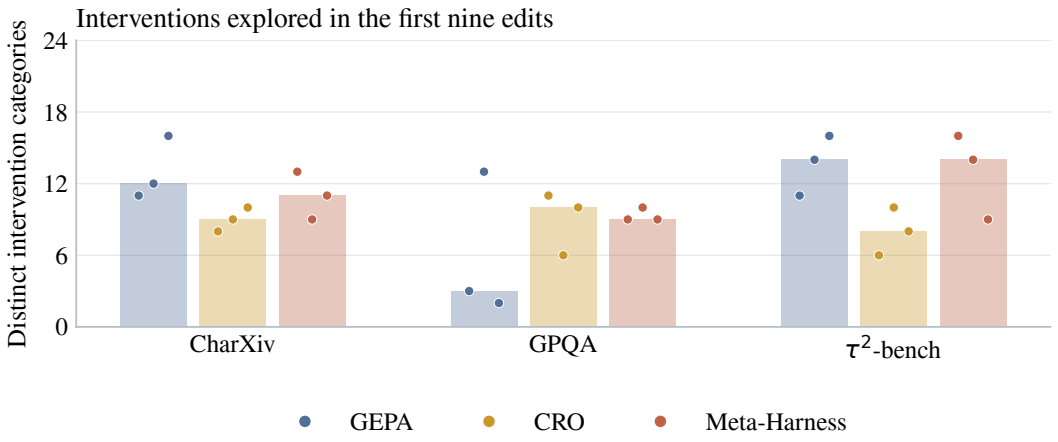

Figure 12: **Interventions explored in the first nine edits.** Distinct intervention categories, excluding those specific to individual items. Each method–task group contains three Opus 5 runs, one per starting tier, using Luna and replicate index zero. Bars are medians; dots are individual runs. Equal edit counts do not imply equal cost or equal numbers of intervention changes. The methods' instructions and permitted edits are documented in Table 7.

The permitted edit is not the same as its implementation. GEPA replaces one declared prompt component while leaving its calling code fixed. In the retained GPQA and CharXiv configurations, CRO can change leaf tasks, add stages, and alter their composition in `Pipeline`. It must preserve the task's entry interface, the imports file, and at least one existing leaf task for caching. It is therefore not restricted to changing one stage per proposal. Meta-Harness permits changes to the worker program within the task and worker contract, including prompts, parsing, control flow, and verification. These scopes determine which implementations are possible, but do not require the proposer to use them.

To inspect actual use, we sampled one parent-relative edit from each of the 27 runs in Figure 12. Sonnet 5 screened these diffs, and we inspected the source for the examples below; we do not treat the automatic classifications as a validated quantitative outcome. A CharXiv GEPA edit adds precision and answer-format guidance to the verifier's prompt. A CRO edit adds a normalization stage that makes an additional model call and connects it to the workflow. A Meta-Harness edit changes a fixed three-call procedure to two calls with conditional escalation. Conversely, a GPQA Meta-Harness edit described as a "verification stage" changes prompt text and comments without adding an execution stage. Thus, an intervention label such as verification does not identify whether an edit changes instructions, the call graph, or a runtime decision.

The distinction is particularly important on τ^2-bench. CRO's workflow returns a prompt template for a fixed dialogue-agent implementation. One sampled edit adds a compiler that parses and reorders prompt blocks: the generator executes new code, but the deployed agent still receives a new prompt. Meta-Harness can instead modify the deployed agent. A sampled edit adds a generation hook that detects tool errors, inserts a repair instruction, and processes JSON-shaped output. This establishes a runtime change, not that the attempted repair succeeds. Neither Python-file counts nor broad intervention families reliably distinguish these cases.

Instructions favor particular uses of the available scope. GEPA explicitly asks for domain-specific facts as well as general strategies. The GPQA prompts accumulate factual reminders, including material from examples the parent already answered correctly. Meta-Harness emphasizes focused edits grounded in failed traces while allowing program changes. CRO asks for alternative interventions, predicted trace changes, and checks against later outcomes; it also suggests sampling, critic/editor loops, and specialized stages when local changes plateau. Some recurring interventions are therefore encouraged by the method's instructions. Their frequency alone cannot distinguish a model's prior preference from the effect of these directions. CRO's existing prediction-and-check

requirement also makes the accuracy of those checks a more informative target for further study than the mere presence of a rationale.

Broader scope does not consistently yield broader exploration. We compare Opus 5 runs with the main worker, replicate index zero, and all three starting-agent tiers: 27 runs across three methods and three tasks. For each run, we count distinct task-level intervention codes touched in its first nine annotated edits, excluding item-specific entries and missing or unresolved codes. This fixes the number of edits, not cost, computation, or the number of interventions changed per edit. The annotations remain provisional. Median breadth for GEPA, CRO, and Meta-Harness is respectively 3, 10, and 9 codes on GPQA; 12, 9, and 11 on CharXiv; and 14, 8, and 14 on τ^2-bench. Even the GPQA ordering varies with the starting agent: from the medium tier, GEPA touches 13 codes and CRO 6. With only three runs per method–task group, these are descriptive comparisons. Instructions, feedback, search policy, and evaluation differ alongside scope. A prompt can express several interventions, and a substantial program change can implement a familiar one; neither intervention breadth nor implementation complexity alone establishes useful diversity or performance gains.

F.4.1 A Controlled Meta-Harness Instruction/Scope Pilot

Design. We cross two instruction conditions with two edit scopes on GPQA and CharXiv, yielding eight trajectories. Each task uses the same minimal starting agent, twelve training items, and a shared initial evaluation and feedback record across its four conditions. Claude Opus 5 proposes edits and GPT-5.6-Luna executes the worker; Luna also judges CharXiv. Neutral instructions retain Meta-Harness's request for focused changes based on failed executions. The suggestion condition additionally names verification/refinement, multi-sample voting, and decomposition as optional interventions. Prompt-only scope permits changes to declared prompt files and requires all other files to match the seed. Program scope additionally permits worker-code changes while preserving the input/output interface and model-call contract.

The pilot uses the Meta-Harness loop with one proposed candidate per iteration, three planned iterations, full training-trace disclosure, and selection on training scores. This small training-selected experiment differs from the main grid's search-selected Meta-Harness configuration. Worker executions permit at most four model calls. Test evaluation uses 24 held-out items with one repetition. Conditions share initial information, but later candidates and feedback diverge. There is one trajectory per task and condition; iterations are dependent observations, not independent replicates. One GPQA suggestion/program iteration ended with a method error, leaving 23 evaluated candidates from 24 attempts. The earlier failed launch is excluded throughout.

Measurement. Frequent-set membership in this completed pilot remains fixed to the original annotation cohort. A blinded Sonnet 5 judge reads each parent source and diff, without the condition, scores, or intervention frequencies. It assigns task-codebook interventions and flags executable changes and item-specific content. These annotations have not received human validation. We count the union of identified intervention codes across the first two evaluated edits, which are iterations one and two in every trajectory. This common horizon was selected after observing the incomplete trajectory; it equalizes edit counts, not cost or the amount of code changed. Codes describe changes in a diff and need not represent newly introduced, useful, or retained interventions.

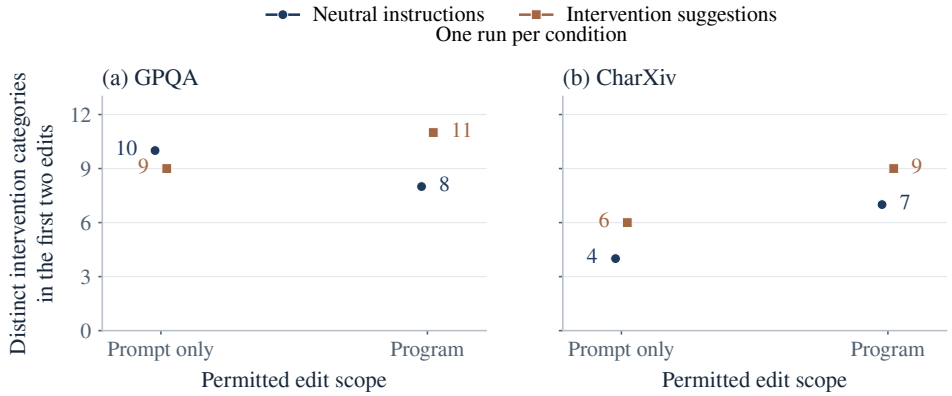

Figure 13: **Intervention coverage under controlled instructions and edit scope.** Each point is one Meta-Harness run's number of distinct intervention categories over its first two evaluated edits. Prompt-only and program-editing conditions are separate runs. All identified categories are counted, including those in edits also marked item-specific. This one-run-per-condition pilot has no estimate of variation between repeated runs.

Results. Program-scope trajectories cover more codes in three of four task–instruction comparisons (Figure 13); the exception is GPQA under neutral instructions. All eight program-scope edits in this equal-horizon comparison are marked as changing executable behavior, and frequent interventions occur in 15 of the 16 edits across scopes. Thus, broader code access is used and can accompany broader intervention coverage, while familiar interventions remain prevalent. Coverage and concentration can coexist because one edit can touch both frequent and less frequent interventions. These observations do not establish a population effect of scope, an instruction-by-scope interaction, or a performance benefit.

Sensitivity to item-specificity filtering. The original pilot summary excluded every code from an edit flagged as item-specific. That flag applies to an entire edit, which can also contain general interventions. In the first two iterations it affects one GPQA neutral/program edit and both CharXiv suggestion/program edits. Their code counts fall from 8 to 6 and from 9 to 0, respectively, under whole-edit exclusion; all other plotted counts are unchanged. The zero therefore indicates discarded mixed edits, not an absence of interventions. We retain their codes in the primary figure and report the alternative explicitly. The result concerns total identified coverage; it does not establish increased coverage of general interventions alone. Intervention-level validation is needed to separate the general and item-specific parts reliably.

Table 23: **All evaluated edits in the instruction/scope pilot.** Three iterations were attempted per trajectory; one produced no evaluated candidate. Common and executable columns count edits, while codes counts distinct identified interventions. Test gains use 24 items with one repetition.

Task	Instructions	Scope	Evaluated	Common	Executable	Codes	Test gain
GPQA	Neutral	Prompt	3	3	0	11	−4.2
GPQA	Neutral	Program	3	3	3	11	+8.3
GPQA	Suggestions	Prompt	3	3	0	12	+4.2
GPQA	Suggestions	Program	2	1	2	11	+4.2
CharXiv	Neutral	Prompt	3	2	0	5	+0.0
CharXiv	Neutral	Program	3	3	3	8	+0.0
CharXiv	Suggestions	Prompt	3	3	0	7	+4.2
CharXiv	Suggestions	Program	3	3	3	11	+4.2

Full-horizon outcomes. Table 23 retains all evaluated edits, including the third iteration where available. The GPQA suggestion/program trajectory has only two, so its full-horizon code count has less opportunity to grow. Twenty-one of 23 edits involve frequent interventions; all eleven program-

scope edits are marked as changing executable behavior. Test gains are exploratory measurements on 24 items, and one item changes accuracy by 4.2 points. The pilot does not support a performance ranking of instructions or scopes.

F.5 Feedback and Proposal Choice

The controlled probe retains its original frequency groups; its 72% concentration result is not recomputed using the expanded cohort.

What the retained feedback reveals. We inspect the information available to the proposer, rather than infer its use from a method's interface. Among retained GPQA GEPA prompts with corresponding edits, 20,790 of 20,824 displayed example outputs contain only an option letter. These are example appearances, including repeated appearances across proposal steps, not distinct questions. The question and reference answer remain visible, but the intermediate computation is absent. A wrong option can suggest several explanations—misreading a qualifier, applying the wrong scientific principle, or making a calculation error—without distinguishing among them. An intervention can consequently be relevant to the question without being established as a remedy for the error that occurred.

Following evidence into the edit. We sampled up to four runs per method and task on CharXiv and GPQA, then one eligible edit per run, without selecting on its measured effect (31 cases; only three eligible CRO GPQA runs). We inspected proposer inputs, completed tool outputs, and code differences. The cases illustrate the distinction between available feedback, observed behavior, and a proposed explanation; they do not estimate comparative success rates. In a Meta-Harness CharXiv session, the inspected intermediate outputs showed correct reader answers being overwritten by a verification stage. The corresponding edit narrowed that stage's trigger and restricted its replacement behavior. The evidence thus suggested when an existing intervention should apply, rather than simply motivating more verification. In a GPQA session, Meta-Harness inspected reasoning that selected an overly inclusive answer and added a check on whether the question's qualifier permits it. By contrast, a CRO proposal permuted option order to test a suspected position effect: the inspected critic outputs motivated a hypothesis, but did not themselves demonstrate that position caused the error. An exploratory proposal need not have an already-proven explanation; its predicted effect must subsequently be tested.

Some retained Sol coding-agent traces contain tool-event placeholders with empty arguments and outputs, and some proposer inputs are unresolved blob references. These cases have unknown evidence exposure. They must not be counted as failures to read feedback. Multiple sibling edits can also share one proposer session, preventing a simple count of session-level reads from establishing what was read before each individual edit. Automated case assessments were used to locate examples, then checked against the records; they were not treated as validated labels for method rankings.

Table 24: **Frequent-intervention concentration by method.** Median percentage of a run's annotated edits touching at least one frequent intervention; run counts in parentheses. The frequent set is fixed within each task using the pooled run cohort, as in Figure 4. These descriptive differences do not isolate feedback from instructions, edit scope, or search policy.

Method	CharXiv	GPQA	τ^2-bench
AdaEvolve	70.0 (16)	62.6 (12)	70.6 (17)
GEPA	99.4 (18)	99.7 (18)	80.4 (18)
Meta-Harness	89.7 (16)	80.5 (18)	84.6 (17)
CRO	72.7 (15)	48.2 (11)	51.6 (9)

Implication for mode collapse. The pooled concentration does not mean every method spends nearly all of its edits on frequent interventions. GEPA is especially concentrated, whereas the code-editing methods differ by task (Table 24). Nor does a frequent intervention necessarily reflect an unsupported choice: execution evidence can motivate a precise refinement within that intervention. The failure analysis therefore needs both the intervention's frequency and the connection between

the observed failure and the specific proposed change. These observations motivate a controlled test of feedback content, with proposer instructions and edit scope held fixed.

A controlled probe of feedback content. We sample one eligible parent state from each of 18 GPQA GEPA runs with the main worker model. Each state has at least four distinct failed training questions recorded for the same parent code hash and prompt; prompts longer than 8,000 characters are excluded. Four failed questions are randomly partitioned into two batches of two, without selecting for a particular diagnosis or inspecting subsequent proposals. GPT-5.6-Sol proposes one primary intervention and a corresponding instruction rewrite under three conditions: aggregate training scores only, or those same scores plus failure batch A or B. The score summarizes the same deduplicated pool of retained training observations in every condition. Parent, task, instructions, edit scope, temperature, and output budget are fixed. Two independent samples per condition yield 108 proposals. This is a standardized one-step prompt-revision probe, not a rerun of native GEPA or its full optimization loop.

A separate model, Claude Sonnet 5, maps each change to at most two intervention codes and assesses its support from both failure batches. The judge is not told the generation condition or which codes are frequent; batch order is randomized. One unavailable judgment excludes its entire parent state from paired summaries, leaving 17 states and 102 proposals. These new assessments have not received the human validation reported for the earlier evidence-justification experiment. We therefore treat them as exploratory annotations, retain unknown support separately, and make no claim about downstream accuracy.

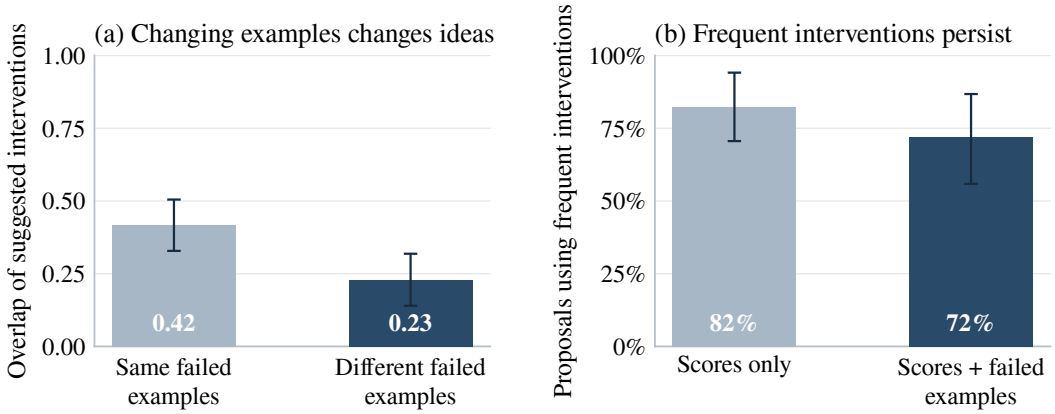

Figure 14: **Feedback changes proposals while frequent interventions remain prevalent.** Exploratory GPQA prompt-revision probe with GPT-5.6-Sol; 17 complete parent states. (a) Mean Jaccard overlap between intervention-code sets from two proposals given the same failure batch, versus proposals given different batches; pairwise overlaps are averaged within each parent before averaging across parents. (b) Share of proposals containing at least one intervention from the existing pooled GPQA frequent set: 28/34 with scores only and 49/68 with failed examples. Whiskers: 95% bootstrap intervals from 5,000 resamples of parent states. The study measures proposal content, not improvement in agent performance.

Changing the failed examples reduces intervention overlap from 0.42 under repeated sampling of the same batch to 0.23 across batches. The paired difference is 0.19 (95% bootstrap interval 0.07–0.31). Thus, the supplied examples affect intervention choice beyond sampling variation in this probe. Nevertheless, 72% of proposals with failed examples still include a frequent intervention. Their frequency is lower than with scores alone, but the paired difference is uncertain (-10 percentage points, interval -28 to $+7$); this sample does not establish that feedback reduces concentration.

The judge marks specific support for 35/68 proposals against their shown batch and 7/68 against the alternative batch; the paired difference is 41 percentage points (interval 22–59). Against shown evidence, 15/68 are marked merely plausible and 18/68 unknown. Score-only proposals receive specific support in 2/68 assessments against the two batches, with 7/68 merely plausible and 59/68

unknown. These judgments assess a proposed connection to the available examples, not whether the intervention would actually prevent the error. In particular, matching an intervention to a question's subject matter need not identify the worker's unobserved reasoning failure. The probe supports responsiveness to feedback alongside continued concentration, rather than a claim that the optimizer universally ignores feedback.

Illustrative change in proposal content. For one fixed parent agent, a failure batch containing an incorrect "all of the above" answer prompted checks on question qualifiers and individual answer-option claims. A different batch, containing synthesis failures, prompted explicit tracking of atoms and bonds through chemical reactions. This example illustrates how the failed questions redirect the proposed intervention; neither proposal was evaluated for whether it repaired the failure.

Directions suggested by the proposal analysis. The findings motivate testing proposers trained to generate distinct, task-relevant interventions, methods that compare competing explanations of a failure, and procedures that allow evidence gathering without requiring every iteration to produce an edit. None of these remedies is validated by the present probe. One hypothesis is that familiar proposals reflect techniques especially useful for earlier, weaker models, which could help explain why weaker base models retain more gain (Figure 3). The comparisons here do not directly test that explanation.

F.6 Retrying Rejected Ideas

A rejected edit does not establish that every intervention it contains is ineffective. A later attempt may change the implementation, use a different parent, or receive a less noisy evaluation. The question is whether a run repeatedly spends effort on interventions that it has not yet retained, and whether those attempts yield improvements.

Definition and measurement. We follow each intervention's non-removal attempts up to its first appearance in a kept edit. Its first attempt is new; subsequent attempts before that first retention are retries. Among these not-yet-retained interventions, an edit is *new-only* if all attempts are new and *retry-only* if all are retries. Either class may also refine interventions retained earlier; the names do not describe every change in the edit. Edits containing both new attempts and retries form a third group. This identity-dependent analysis retains the first annotation batch and requires at least nine annotated edits; late-batch results are reported separately in Table 15. Effort shares use edits with at least one not-yet-retained intervention and their attributed optimizer costs, excluding worker execution costs.

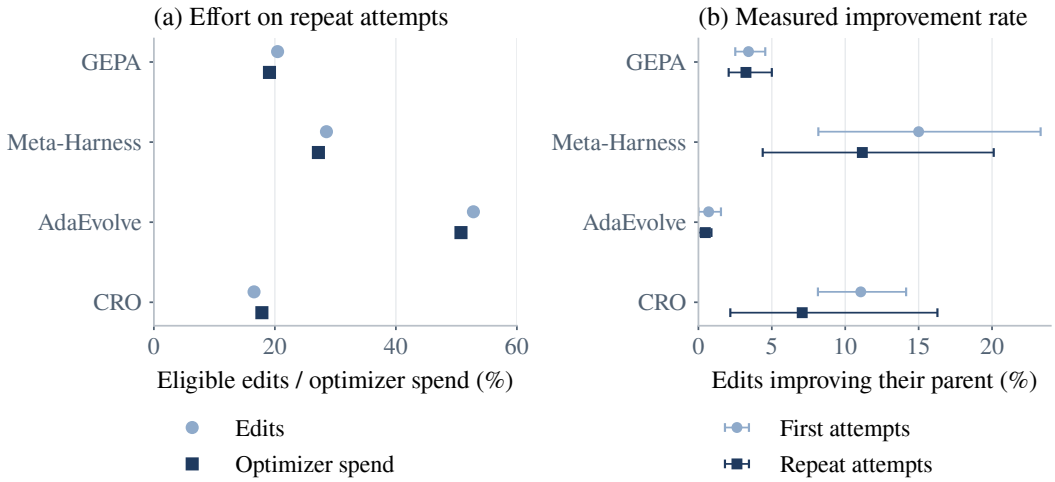

Figure 15: **Retry effort and measured yield in the first batch.** (a) Retry-only shares of eligible edits and optimizer spend, pooled within method. (b) Share of measured new-only and retry-only edits that improve their parent under the one-standard-error rule. Whiskers: 95% intervals from 10,000 whole-run bootstrap draws within method. The two groups need not contain the same runs or equally strong parents; these are descriptive comparisons. In the plots, "First attempts and "Repeat attempts concern only interventions not yet kept; either group can also refine already-retained interventions.

Results and interpretation. Retry-only edits account for 17–53% of eligible edits across methods and a similar fraction of optimizer spend (Figure 15a). Their measured improvement rates are lower than new-only edits' point estimates for every method, but the comparison does not match parent quality, timing, or intervention difficulty. Interventions that work early leave the not-yet-retained group, making later retries a selected, harder population. In the existing 114-run comparison, retry share has little association with search or test gain (within-method rank correlations $+0.03$ and $+0.04$, adjusted for run length). Only six retry-only promotions have paired test measurements, insufficient to characterize their held-out benefit.

Implication for memory. A useful history should record what was tried, on which parent, what the evaluation could resolve, and what changed before a retry. It should not simply prohibit previously rejected interventions. An exploratory replay that skips every edit whose not-yet-retained interventions have already failed once would skip 17% of edits and 19% of optimizer spend, but also discard 59 of 476 measured improvements and 52 of 413 promotions. Waiting for two failures reduces the skipped shares to 8% each, while discarding 26 improvements and 18 promotions. These are deletions from a fixed recorded sequence, not simulated alternative searches or demonstrated cost savings. They motivate evidence-aware retries rather than a blanket rejection memory.

F.7 Rapidly Diminishing Returns

Yield over the expanded run cohort. Figure 16a uses all measured edits in the expanded Luna cohort, without relying on intervention identity. We divide runs into five spans of normalized edit position, pool edits within method and span, then average the available method rates equally (at least 20 measured edits per method/span). The late batch has no GEPA runs, so its composition differs. Normalized position is retrospective: final run length would not be known to an online stopping rule.

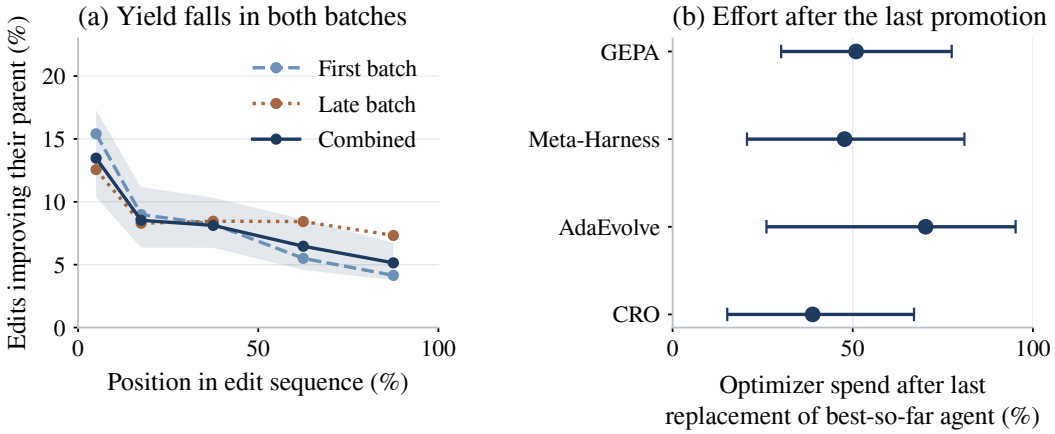

Figure 16: **Declining edit yield in both annotation batches.** (a) Measured improvement rates in five spans, shown separately for the first batch, late batch, and pooled Luna cohort. Methods are weighted equally within each series; no identity-dependent new/retry categories are pooled. Lines connect descriptive rates; the shaded pooled band is a pointwise 95% interval from 10,000 whole-run bootstrap draws within method. (b) Median optimizer spend after the final recorded promotion in the expanded cohort with at least nine edits; whiskers span the middle half of runs. Runs without a promotion contribute 100%. These are recorded selection outcomes, not independently confirmed gains.

Across the pooled cohort, measured improvement rates decline from 13.5% in the first tenth to 5.1% in the final quarter; corresponding rates are 15.4% to 4.1% in the first batch and 12.6% to 7.3% in the late batch. In the separate first-batch identity analysis, new-only edits decline from 15.4% to 4.7%, so retries alone do not explain that batch's decline. Later parents may have less headroom, remaining ideas may be harder, and score noise may hide small gains; these observations do not isolate the explanation.

Budget and stopping. Median optimizer spend after the last promotion ranges from 39% for CRO to 70% for AdaEvolve (Figure 16b). That fraction is not automatically wasted: later experiments can reject alternatives or develop a branch without producing another best score. The final promotion is also identifiable only after the run. A prospective stopping policy would need to estimate the probability and value of another confirmed gain and compare it with moving effort to another branch or task; simply stopping at the retrospective last promotion is unavailable.

Returning to earlier parents. In an exploratory CRO comparison after at least two rejections, 25% of 95 edits built on the seed are kept, versus 11% of 290 built on the incumbent. This suggests a possible role for revisiting earlier starting points, but seed and incumbent edits differ in headroom, timing, and selection context. It does not establish that restarts improve held-out performance. Appendix H.2 examines these parent-choice limitations.

F.8 Full Inventory of Frequent Interventions

Tables 25–30 give the codebook description and run prevalence of every frequent intervention. An intervention is included when at least half of the task's eligible runs propose it. The cohorts contain at least nine annotated edits per run and pool optimizer models; they therefore differ from the optimizer-specific sets in Table 20. Prevalence measures proposals, not retention or demonstrated benefit. No-feedback coverage is reported separately in the optimizer-matched comparison (Appendix F.2).

Table 25: **Frequent interventions on CharXiv.** 65 eligible Luna runs; at least half propose each listed code.

Intervention	Description	Runs (%)
Question-type router	Classify the question into a taxonomy of types (count, extremum, label, arithmetic, threshold, trend, etc.) via regex or LLM and dispatch to type-specific prompts or procedures.	86
Structured step-by-step reading procedure	Replace free-form instructions with a numbered multi-step protocol covering axes inspection, legend matching, candidate enumeration, arithmetic, and answer formatting.	83
Two-pass draft-then-verify	Run a first LLM call to produce a draft answer, then run a second independent LLM call that audits and corrects the draft before returning the final answer.	65
Exhaustive candidate enumeration before conclusion	Require the model to list every candidate (series, panel, category, tick) with its measured value before applying a superlative, count, or threshold criterion.	63
Tick-level precision snapping	Instruct the model to round numeric answers to the nearest labeled gridline or tick value and forbid extra significant digits from pixel-level interpolation.	63
Answer post-processing normalizer	Apply a deterministic or LLM-based post-processing stage that strips markdown artifacts, units, prefixes, and formatting wrappers from the extracted answer string.	63
Falsification / adversarial pass	Add a dedicated pass that attempts to disprove the draft answer by finding counterexamples or alternative readings before committing to the final answer.	63

Table 26: **Frequent interventions on CharXiv.** 65 eligible Luna runs; at least half propose each listed code. Continued.

Intervention	Description	Runs (%)
Structured evidence inventory before answering	Require the model to produce a structured evidence record (TARGET, CANDIDATES, VALUES, COMPUTE) before committing to a final answer.	62
Bare minimal answer format	Require the final Answer line to contain only the bare number, label, or short phrase with no units, explanations, LaTeX, hedging words, or trailing text unless explicitly requested.	58
Verbatim label copying	Require character-for-character reproduction of axis labels, legend entries, panel identifiers, and category names exactly as printed, including capitalization, punctuation, and special characters.	57
Dedicated arithmetic execution	Route arithmetic (differences, sums, ratios) through a Python code block or sandbox execution rather than relying on the model's stated mental arithmetic.	55
Unknown/Not-Applicable fallback rule	Instruct the model to output 'Unknown' or 'Not Applicable' when the chart does not support the question's premise, rather than hallucinating a plausible answer.	54
Counting enumeration with recount	Require element-by-element enumeration in a fixed scan order followed by a reverse-direction recount to catch off-by-one errors.	51
Diverse-role multi-pass pipeline	Run multiple LLM calls with different role framings (e.g., DIRECT, TRANSCRIBE, SKEPTIC or DRAFT, CORRECT, SELECT) and adjudicate among their outputs.	51

Table 27: **Frequent interventions on GPQA.** 59 eligible Luna runs; at least half propose each listed code.

Intervention	Description	Runs (%)
Structured solving procedure in initial prompt	Replace a minimal or terse instruction with a numbered multi-step derivation procedure (restate domain, derive from first principles, check units/signs, eliminate distractors, sanity-check) in the initial/system prompt.	69
Structured audit checklist in critique prompt	Replace a generic critique instruction with a numbered audit checklist (re-read stem, independent calculation, principle verification, option mapping, etc.).	64
Adversarial falsifier / challenger pass	Add a pass that challenges or tries to falsify the draft answer, either by steelmanning rivals or attempting to disprove the consensus, before finalizing.	59
Multiple independent solver calls with majority vote	Run N independent LLM calls (with varied prompts, temperatures, or reasoning styles) and select the answer by majority or plurality vote.	58
Domain-specific knowledge bullets in prompt	Inject reusable domain-principle reminders (formulas, rules, scaling laws) as bullet points into the solver or revision prompt, covering multiple subject areas.	54
Blind independent re-solve in evaluator	Instruct the evaluator or critic to re-derive the answer independently from first principles, ignoring the majority vote or branch reasoning.	54
Multi-pattern regex answer extraction	Use a tiered regex cascade to extract the answer letter, preferring a terminal 'Answer:(X)' pattern, then parenthesized letter, then bare letter.	53

Table 28: **Frequent interventions on GPQA.** 59 eligible Luna runs; at least half propose each listed code. Continued.

Intervention	Description	Runs (%)
Item-specific gold-answer calibration facts	Embed specific correct answers and their reasoning for known training items directly into the prompt as calibration or benchmark bullets.	51
Constraint ledger / qualifier enumeration	Before solving, enumerate all stated conditions, qualifiers, and MUST-CONSUME constraints from the question stem and verify the chosen option satisfies all of them.	51

Table 29: **Frequent interventions on τ^2-bench.** 61 eligible Luna runs; at least half propose each listed code.

Intervention	Description	Runs (%)
Comprehensive system prompt structure	Replace terse operating rules with a comprehensive multi-section structured policy covering authentication, tool use, confirmation, retail rules, airline rules, payment, communication style, and error handling.	67
Single confirmation, execute immediately	Ask for confirmation exactly once; execute immediately on any affirmative reply; re-asking the same confirmation is a scored failure.	66
Pre-write argument verification	Before every write/mutation tool call, silently re-verify all arguments (IDs, statuses, amounts, payment methods) against the latest tool output.	62
Pre-send obligation/compliance walk	Before sending any message, silently enumerate all obligations or sub-requests from the conversation and verify each is addressed or executed.	61
Atomic dependency-ordered execution	After a single confirmation, execute all approved writes atomically in dependency order without pausing between steps.	56
Plain prose output format	Require all customer-facing replies to be plain conversational prose; forbid JSON objects, code fences, key-value dumps, or structured blobs in output.	54
Retry after tool error with repair	When a tool returns an error, intercept any prose-only response and retry once with a targeted repair; accept the revision only if it contains a corrected tool call.	52

Table 30: **Frequent interventions on τ^2-bench.** 61 eligible Luna runs; at least half propose each listed code. Continued.

Intervention	Description	Runs (%)
Structured per-turn execution ledger	Maintain a private per-turn ledger tracking each user request with status (OPEN/DONE/BLOCKED), prerequisites, and authorization; reconcile after every tool result.	52

G Evaluation: Further Results

This appendix distinguishes three questions: how selection favors high measurement errors, whether recorded improvements transfer to withheld items, and how a stored score affects subsequent comparisons. A search–test difference mixes execution noise and transfer; it is not a direct measurement of either component alone.

G.1 Repetition Budget and Selection Reliability

Question and model. Figure 5b–c asks how repeated evaluation changes the ability to distinguish two fixed candidates on the same search examples. Let their expected scores differ by $\Delta > 0$, and let each candidate's one-repetition score have execution-noise standard deviation σ, in percentage points. Both candidates receive k independent executions per example. Under independent, unbiased Gaussian errors with equal variance, their measured score difference is

$$\widehat{S}_k(h_1) - \widehat{S}_k(h_2) \sim \mathcal{N}\left(\Delta, \frac{2\sigma^2}{k}\right), \qquad P(\text{choose better}) = \Phi\left(\frac{\Delta\sqrt{k}}{\sqrt{2}\sigma}\right), \tag{3}$$

where Φ is the standard Normal cumulative distribution function. Panel (b) shows $\sigma/\sqrt{k}$, the standard deviation of one candidate's measured score; the comparison in panel (c) includes uncertainty in both scores. These probabilities concern which candidate has higher expected score on the fixed search items, not which generalizes better to test.

The square-root rule and evaluation cost. Averaging k independent executions divides variance by k and standard deviation by $\sqrt{k}$. To achieve a specified pairwise selection probability $p > 1/2$, the model requires

$$k \geq \frac{2\sigma^2[\Phi^{-1}(p)]^2}{\Delta^2}. \tag{4}$$

Thus, halving the expected-score gap requires four times as many repetitions for the same decision reliability; doubling evaluation noise also requires four times as many. With a fixed number of examples, these multipliers apply to execution counts for both candidates, not necessarily to dollar cost across different programs. A small absolute noise level can still obscure an even smaller gain. Conversely, a larger gain is easier to distinguish at the same noise level. These relationships motivate allocating evaluation effort according to the comparisons that need resolving; they do not establish the benefit of any particular adaptive stopping or confirmation policy.

Noise calibration. We use the available Luna/OpenAI calibration settings from Appendix J: five for GPQA, six for CharXiv, and five for τ^2-bench, including the documented exploratory close-parent fits. Their fixed search sets contain 67, 66, and 55 items, respectively. For each calibration seed, we multiply the estimated variance of its averaged score by its recorded repetition count before taking the square root to recover a one-repetition noise estimate. All 16 calibration seeds used three repetitions. We then take the median across settings within each task. This gives $\sigma = 3.11$, 3.27, and 5.35 points. These seed-based estimates approximate noise for edited candidates; they are distinct from the fresh endpoint evaluations in Appendix G.2. The model holds the item set and execution distribution fixed as repetition count changes.

Estimating a typical score advantage. The two-parent fits separate variation in expected child scores from evaluation noise, under the assumptions in Appendix J. If independent proposals from the same parent have normally distributed expected scores with standard deviation s, their median absolute expected-score difference is $\sqrt{2}\Phi^{-1}(0.75)s \approx 0.954s$. We apply this conversion to each setting's posterior median proposal spread, take the median across available settings within each task, and round to one decimal. The resulting gaps are 0.7, 1.6, and 1.9 points. These are model-implied summaries, not independently verified true advantages or the gaps between the best two candidates in a search. Using separately summarized gaps and noise defines a representative scenario; it does not estimate the average selection success rate across methods or runs.

Table 31: **Calibration and projected pairwise selection probabilities for Figure 5b–c.** Noise and expected-score gaps are in percentage points. Probabilities condition on the displayed task summaries; uncertainty in their estimation is not propagated.

	One-rep.	Expected	Probability of choosing better (%)			
Task	noise σ	gap Δ	1 rep.	3 reps.	5 reps.	6 reps.
GPQA	3.11	0.7	56.3	60.9	63.9	65.2
CharXiv	3.27	1.6	63.5	72.5	78.0	80.2
τ^2-bench	5.35	1.9	59.9	66.8	71.3	73.1

Scope of the projection. Both curves follow the stated model, calibrated to existing measurements; no new selection experiment was run. The assumed expected scores remain fixed as repetitions increase, and no observed six-repetition score is treated as noise-free ground truth. The estimates omit uncertainty in proposal spread and evaluation noise, dependence between executions, serving drift, and item-sampling uncertainty. The two-candidate comparison also omits the favorable error inherited by a previously selected incumbent and the difficulty of choosing among many candidates. It therefore illustrates the cost of resolving small performance differences, not the magnitude of winner's curse or the probability that an entire optimization run returns its truly best candidate. The numeric inputs and reproducible calculation are provided in `analysis/independent_validation/analysis/selection_probability.json` and `analysis/plot_readability/evaluation_selection.py`.

G.2 Independent Reevaluation of Returned Agents

Question and frozen cohort. We ask whether a returned program's reported improvement survives independent executions on the search items, and whether it improves on held-out items. The cohort is fixed before observing fresh outcomes: GPQA and CharXiv, GEPA/Meta-Harness/AdaEvolve, and Opus/Sol optimizers, with Luna as worker, the minimal starting agent, and optimization seed 0 in all 12 conditions. Optimizer models are separate strata, not random-seed replicates. The study excludes CRO, other starting tiers, and weaker workers; it does not represent every setting in the main grid.

Programs and measurements. We freeze each run's actual initial and returned programs and preserve its original item IDs: 67 GPQA search items, 66 CharXiv search items, and 66 test items per task. The initial program is recovered from the first segment for the resumed Opus/CharXiv/Meta-Harness run; AdaEvolve's initial program retains its method-specific conversion. We collect six fresh executions per item after selection, using seeds 980000–980005, excluding the earlier pilot. Identical starting programs with identical item sets share evaluations, yielding 32 distinct program/split units, 192 complete repetitions, and 12,720 rollouts. The worker is `gpt-5.6-luna`; the CharXiv judge is `gpt-5.4-mini`. Search and test gains both subtract the freshly measured starting program. Historical search gain uses the returned program's recorded search score minus its recorded initial score, rather than the maximum score of a different candidate. This matters because AdaEvolve selects using training performance. No fresh outcomes are used to choose programs or conditions.

Uncertainty. For unit u, let Y_{uir} be the score on fixed item i in fresh repetition r, with n_u items and $R = 6$ repetitions. We estimate its mean and variance of the mean as

$$\bar{Y}_u = \frac{1}{n_u R} \sum_{i,r} Y_{uir}, \qquad \widehat{V}_u = \frac{1}{n_u^2 R} \sum_i s_{ui}^2, \tag{5}$$

where s_{ui}^2 is the sample variance across repetitions of item i. An aggregate gain is $100 \sum_u w_u \bar{Y}_u$, where each condition contributes equal weight to its returned program and the negative of that weight to its initial program. Its estimated variance is $100^2 \sum_u w_u^2 \widehat{V}_u$. Accumulating weights for shared programs before squaring retains the covariance induced by reused starting measurements. Search–test differences use positive search weights and negative test weights. We report approximate normal 95% intervals. They condition on these programs and items and assume independent execution draws with comparable measurement conditions; they do not quantify uncertainty across new optimizer runs, tasks, or serving changes. Subgroup intervals are descriptive and not adjusted for multiple comparisons. Historical readings are fixed comparison values, not noise-free estimates.

Table 32: **Freshly measured search gains need not transfer.** Mean gains in percentage points over the same starting programs, six conditions per task. Recorded search gain uses the returned program, not the peak score of another candidate. Fresh-measurement intervals are reported in Table 33.

Task	Recorded search	Fresh search	Fresh test
GPQA	5.39	1.82	2.02
CharXiv	3.11	3.79	-0.04

Table 33: **Independent gains by task.** Equal-weight averages over six conditions per task. Gains and interval endpoints are percentage points; difference is fresh search minus fresh test. Intervals describe execution uncertainty on the fixed programs and items.

Task	Fresh search [95%]	Fresh test [95%]	Difference [95%]
GPQA	$1.82\ [-0.33, 3.98]$	$2.02\ [0.30, 3.74]$	$-0.20\ [-2.95, 2.56]$
CharXiv	$3.79\ [1.81, 5.76]$	$-0.04\ [-2.04, 1.96]$	$3.83\ [1.02, 6.64]$
Combined	$2.81\ [1.35, 4.27]$	$0.99\ [-0.33, 2.31]$	$1.82\ [-0.15, 3.78]$

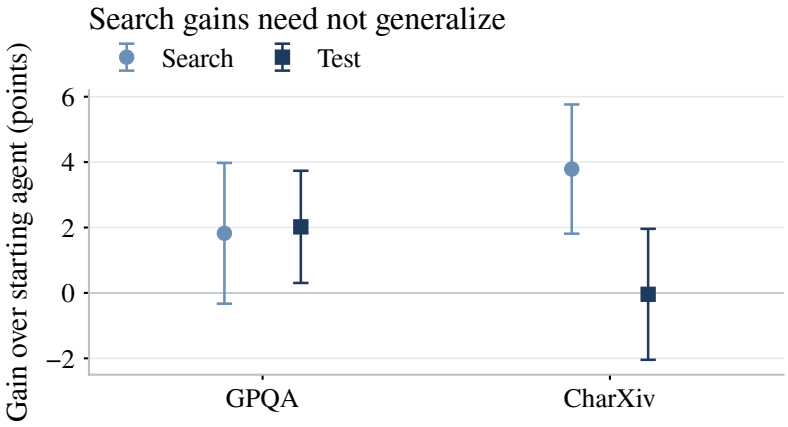

Figure 17: **Search gains can persist without test gains.** Gains of frozen returned agents over their starting agents, six conditions per task and six fresh repetitions per item. Error bars show 95% execution-uncertainty intervals conditional on fixed programs and items. CharXiv retains search gains but has approximately zero mean test gain; GPQA retains test gains.

Results. On GPQA, all six recorded search gains fall under reevaluation: their mean changes from 5.39 to 1.82 points. Fresh test gain remains 2.02 points, consistent with useful improvement. On CharXiv, fresh search gain is 3.79 points but test gain is -0.04; every condition has a larger fresh search than test gain. Their average difference is 3.83 points, with interval $[1.02, 6.64]$. Thus, confirming a gain on the search items does not ensure transfer to the held-out split. Across both tasks, the fresh difference is 1.82 points with interval $[-0.15, 3.78]$; the pooled comparison alone is less conclusive. Nine historical search gains shrink and three CharXiv gains increase, so the historical-to-fresh change is not uniform.

Table 34: **All 12 frozen conditions.** Gains over the corresponding starting program, in percentage points. Fresh gains include approximate 95% fixed-item intervals. Each row has six repetitions; shared starting-program evaluations are reused.

Method	Optimizer	Recorded search	Fresh search [95%]	Fresh test [95%]
GPQA				
GEPA	Opus	4.48	1.49 [−1.94, 4.93]	1.77 [−1.06, 4.59]
Meta-Harness	Opus	8.46	3.48 [0.02, 6.94]	1.77 [−1.14, 4.68]
AdaEvolve	Opus	6.47	1.49 [−1.77, 4.76]	3.28 [0.62, 5.95]
GEPA	Sol	2.49	−1.49 [−5.10, 2.12]	0.25 [−2.76, 3.26]
Meta-Harness	Sol	5.47	4.73 [1.37, 8.08]	2.27 [−0.50, 5.05]
AdaEvolve	Sol	4.98	1.24 [−2.17, 4.66]	2.78 [−0.01, 5.57]
CharXiv				
GEPA	Opus	5.56	3.28 [0.18, 6.39]	−1.52 [−4.94, 1.91]
Meta-Harness	Opus	9.60	5.81 [2.47, 9.14]	3.28 [−0.10, 6.66]
AdaEvolve	Opus	-1.52	3.79 [0.38, 7.20]	2.02 [−0.98, 5.02]
GEPA	Sol	3.03	−0.76 [−4.09, 2.58]	−3.54 [−6.94, −0.13]
Meta-Harness	Sol	0.00	6.06 [2.82, 9.30]	1.01 [−2.32, 4.34]
AdaEvolve	Sol	2.02	4.55 [1.29, 7.80]	−1.52 [−4.58, 1.55]

Serving-route sensitivity. Collection began through the direct API route and resumed through AUDIT LiteLLM using the same requested worker and judge model identifiers. The first four repetitions used the direct route for every unit (128 evaluations); repetition five straddled the switch (13 direct and 19 AUDIT); repetition six used AUDIT throughout (32). Thus 141 evaluations used the direct route and 51 used AUDIT. Because route and collection time are confounded, pooled intervals cannot capture a systematic serving shift. Table 36 repeats the analysis using only the four complete pre-switch repetitions. GPQA again retains test gains; CharXiv again has much larger search than

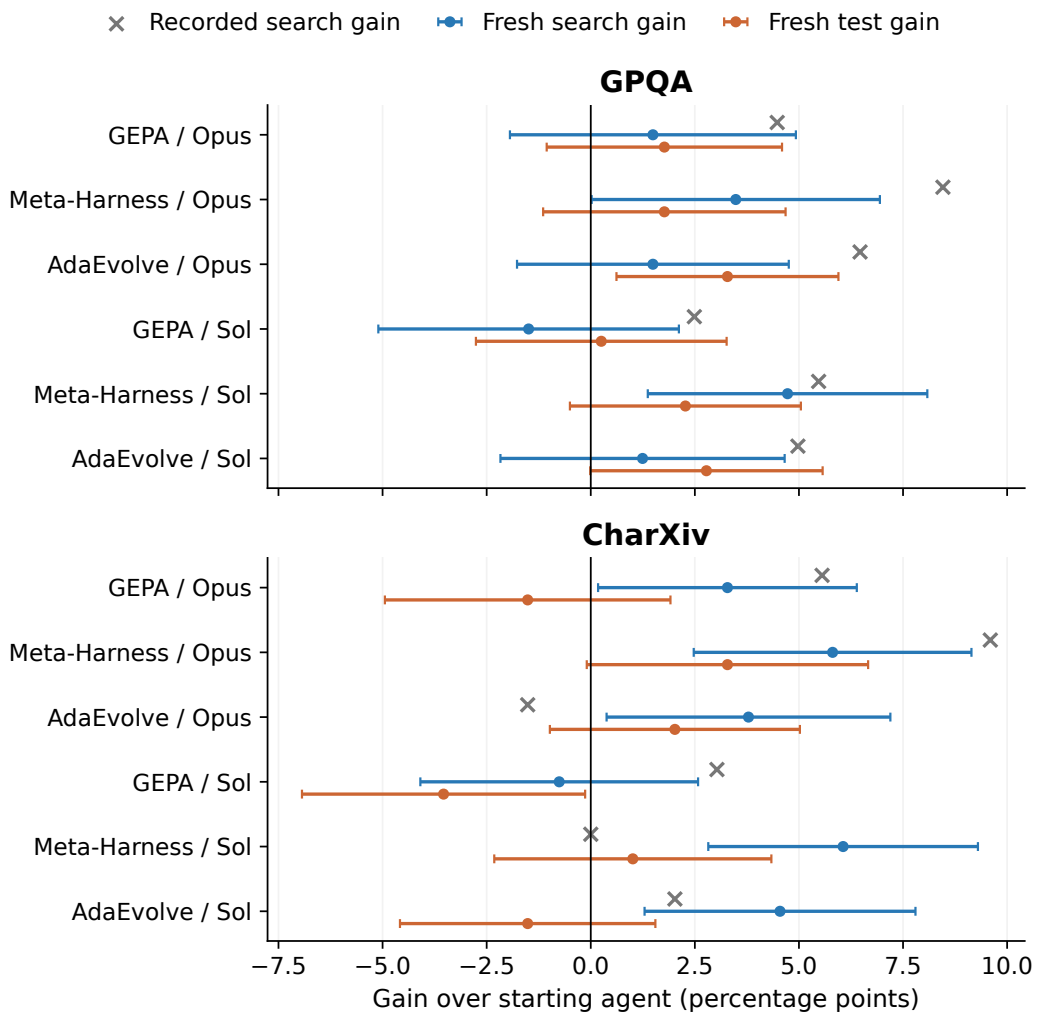

Figure 18: **Recorded and independently measured gains for every condition.** Crosses mark historical search gains; circles mark fresh search and test gains with 95% fixed-item intervals. All gains compare the returned program with its starting program. Historical-to-fresh differences include possible serving changes and cannot be identified entirely as selection noise.

Table 35: **Method and optimizer summaries within the frozen cohort.** n counts conditions, not independently sampled optimization seeds. Gains are percentage points. These descriptive groups do not establish a general method ranking.

Group	n	Recorded search	Fresh search [95%]	Fresh test [95%]
GEPA	4	3.89	0.63 $[-1.46, 2.73]$	-0.76 $[-2.70, 1.18]$
Meta-Harness	4	5.88	5.02 $[2.93, 7.10]$	2.08 $[0.17, 3.99]$
AdaEvolve	4	2.99	2.77 $[0.69, 4.84]$	1.64 $[-0.17, 3.45]$
Opus optimizer	6	5.51	3.22 $[1.63, 4.82]$	1.77 $[0.32, 3.22]$
Sol optimizer	6	3.00	2.39 $[0.78, 3.99]$	0.21 $[-1.25, 1.67]$

test gains. The single complete AUDIT repetition gives pooled search/test gains of $5.01/-0.38$ points, but is insufficient to identify a route effect separately from execution variation.

Table 36: **Sensitivity using only the direct API route.** Four fresh repetitions for every program/split unit, before the route switch. Gains and 95% fixed-item intervals are in percentage points. Difference is search minus test.

Task	Fresh search [95%]	Fresh test [95%]	Difference [95%]
GPQA	$1.99\ [-0.56, 4.54]$	$2.84\ [0.70, 4.99]$	$-0.85\ [-4.18, 2.48]$
CharXiv	$4.36\ [2.02, 6.69]$	$0.38\ [-2.07, 2.83]$	$3.98\ [0.59, 7.36]$
Combined	$3.17\ [1.44, 4.90]$	$1.61\ [-0.02, 3.24]$	$1.56\ [-0.81, 3.94]$

Sensitivity to item composition. We also resample item IDs within each task and split, using the same sampled IDs across programs and retaining all six observed outcomes per item. In 20,000 draws, the pooled 95% percentile intervals are $[0.74, 4.94]$ for search gain, $[-1.35, 3.35]$ for test gain, and $[-1.28, 4.99]$ for their difference. This exploratory paired-item bootstrap checks sensitivity to the fixed benchmark sample; it does not estimate uncertainty across new tasks or optimizer runs, and its variance is not added to the execution-uncertainty intervals.

Integrity and scope of the inference. All 192 main-study records contain the exact pinned item IDs and six complete repetitions per unit; all 12,720 rollouts have successful exits and received model responses, with zero recorded error responses. Recomputing scores from item outcomes reproduces the archived means and covariance-aware standard errors. Pilot attempts are excluded. The main-study cost estimate is $143.37 from token-price accounting, not a verified provider invoice. The optional frozen-pool reselection study was not run.

The fresh evaluations independently check performance after selection, unlike repetitions already included in a winning score. However, equivalence between historical and fresh serving conditions is unverified. Their difference is therefore a remeasurement discrepancy, not an identified estimate of winner's curse. Initial scores also change, so the discrepancy cannot be attributed entirely to returned-program scores. Fresh search–test differences can reflect adaptation to the reused search examples or differences between the fixed splits. These measurements validate endpoint performance; they do not establish a causal benefit from changing selection, validate intervention annotations, or directly test the convergence forecast. The release includes the frozen manifests, per-item outcomes, and scripts needed to reproduce these comparisons.

G.3 Measured Effects and Selection Noise

Effect threshold. An edit measurably improves or regresses its parent when the item-paired score difference in the existing evaluations is at least one estimated standard error above or below zero. A promotion is compared with the incumbent it replaces under the same rule. The parent and the replaced incumbent can be different candidates. These thresholds describe measurement resolution; they are not independent confirmation of an improvement or a multiple-comparison-adjusted significance test. Among the 150 paired promotions below, the one-standard-error rule gives 33 positive, 106 within-noise, and 11 negative search steps. At 1.96 standard errors, the counts are 4, 145, and 1. The classification therefore depends materially on the threshold.

Winner's curse under a simple model. For T fixed candidates with similar expected scores on the same items and independent Gaussian measurement errors with standard deviation σ, the selected score's expected overstatement is approximately

$$\mathbb{E}[\hat{S}(h^\star) - S(h^\star)] \approx \sigma m_T, \qquad m_T = \mathbb{E}\left[\max_{t \le T} z_t\right], \quad z_t \overset{\text{iid}}{\sim} \mathcal{N}(0,1). \tag{6}$$

Here m_T is about 1.2, 1.5, and 1.9 for 5, 10, and 20 candidates. This illustrates amplification of execution noise by selection. Adaptive proposals, unequal variances, and candidate-quality differences require additional modeling. Appendix K gives the assumptions and same-item remeasurement evidence.

G.4 Allocation of Evaluation Effort

Evaluation allocation. We audit the full evaluation events in the original 137-run annotation snapshot: 54 GEPA, 38 Meta-Harness, 12 AdaEvolve, and 33 CRO runs. A full evaluation covers all items in its split and is neither partial nor a subset screen. The selection set is training for AdaEvolve and search for the other methods. We count distinct candidates by their recorded content hashes, including the seed; different versions with equivalent behavior still count separately. A repeated evaluation is another full evaluation of the same candidate hash on the same split. Preliminary screens and test evaluations are excluded.

Table 37: **Evaluation allocation in the provisional annotation sample.** Candidate counts are medians per run. Repeated-call shares pool full evaluations on each method's selection set; they measure calls, not dollars. Coverage differs across methods: AdaEvolve has only minimal-seed GPQA and CharXiv runs.

Method	Runs	Selection set	Distinct candidates	Attempts per item	Repeated calls / full calls
GEPA	54	Search	59	3	36/4,935 (0.73%)
Meta-Harness	38	Search	27	3	1/1,177 (0.08%)
AdaEvolve	12	Training	224	1	14/2,417 (0.58%)
CRO	33	Search	11	3	18/526 (3.42%)

Repeated calls are an upper bound on deliberate confirmation: they include restarts, repeated proposals, seed remeasurements, and final checks, and need not update the score used for selection. These calls are distinct from the fixed repetitions within an evaluation. In the four settings with traces for all four methods (GPQA and CharXiv, both optimizer models, minimal starting agent, random seed 0), median candidate counts are 210 for GEPA, 27.5 for Meta-Harness, 275 for AdaEvolve, and 13 for CRO. The broad ordering persists in this small matched subset, but neither comparison isolates a causal method effect. Recorded worker spending also includes feedback and some held-out scoring, and cost per evaluation depends on the candidate, so a worker-spend fraction is not a clean measure of effort allocated to selection.

AdaEvolve's selection scores and held-out gains. For each of the 12 AdaEvolve runs, we identify the returned candidate by its content hash and its highest recorded training score, which equals the run's maximum training score. We pair that winning training evaluation with the subsequent search evaluation of the same candidate during that promotion, before any further training evaluation. Thus all three measurements in Table 38 concern the same returned agent. The search measurement is not the highest search score attained by any candidate in the run, which can belong to a different agent or an earlier evaluation.

Table 38: **AdaEvolve's returned agents across evaluation splits.** Mean gains in points, with 95% bootstrap intervals from 20,000 resamples of the 12 runs. Training scores guide selection; search and test scores do not.

Evaluation split	Mean gain	95% interval
Training (selection)	9.52	[7.92, 11.26]
Search	1.37	[−0.22, 2.99]
Test	0.74	[−0.48, 2.00]

Every run's recorded training gain exceeds both its search and test gain. The sample contains six runs per task (GPQA and CharXiv), with two optimizer models and three random seeds, all from the minimal starting agent. Training and search gains subtract the run's initial seed reading on that split; test gains use the pooled seed baseline of §5.1. Training evaluations use one attempt per item and search evaluations three. Different item sets, measurement precision, and baseline precision all affect these comparisons. The gaps therefore combine selection of measurement noise and gains specific to the selection examples; they do not isolate either contribution. The bootstrap describes variation across these runs, without accounting for every shared task or infrastructure dependence.

G.5 Overtrusting Dev Scores

Question and cohort. A development score is useful for guiding a search only to the extent that its apparent improvements survive an appropriate independent check. We examine 150 promotions from 43 GEPA, Meta-Harness, and AdaEvolve runs with item-paired test measurements of every promoted candidate. Resumed runs are excluded because their recorded promotion sequence begins at resumption. The source cohort also excludes broken sealed-test sessions and promotion pairs with absolute test changes of at least 15 points. All retained runs have complete promotion sequences, but the results describe this filtered sample rather than every run or every large change.

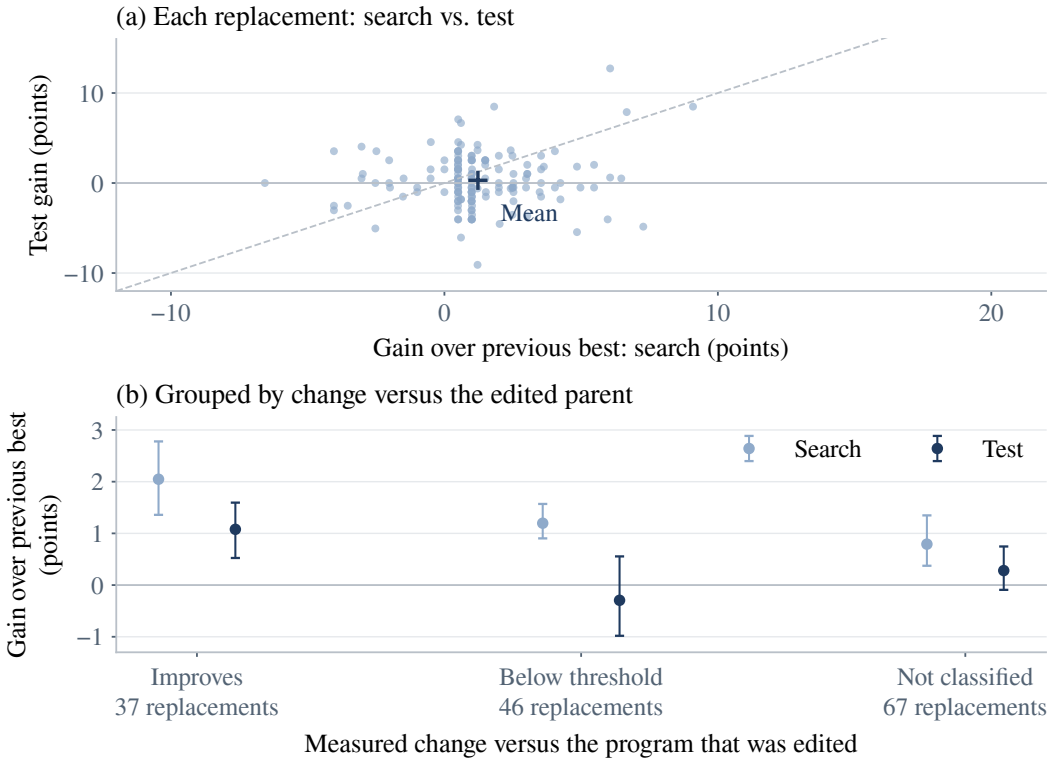

Figure 19: **Recorded promotion gains and held-out changes.** (a) The same 150 promotions measured against the replaced incumbent on search and test; the cross marks the mean and the dashed line equal gains. (b) Mean steps grouped by the promoted edit's measured effect on its own parent. The unavailable group retains all 67 promotions without that classification. Whiskers: 95% intervals from 10,000 whole-run bootstrap samples within group. Groups need not contain the same runs.

What survives on test. The average step is 1.23 points on search and 0.30 on test. The latter's run-bootstrap interval is $[0.02, 0.65]$; step sizes on the two splits have almost no rank association (-0.02). This is evidence that recorded search progress gives limited information about an individual promotion's transfer. It does not distinguish selection of execution noise from adaptation to the reused search items. Figure 19b adds a related diagnostic: promotions whose edits measurably improve their own parents have larger mean test gains than those within noise of their parents. The groups are observational, and using the same recorded evaluations to form them does not validate a new acceptance rule.

Method-specific selection. A promotion's search step exceeds its test step by 0.19 points in AdaEvolve, 1.02 in GEPA, and 1.58 in Meta-Harness. Those numbers cannot rank selection reliability: AdaEvolve chooses on training scores, whereas the other two use search scores. Table 38 compares the same AdaEvolve returned agents on all three splits and reveals the larger discrepancy on the scores that actually guide selection. An evaluation audit must therefore identify which score the method selects on before interpreting a small held-out gap.

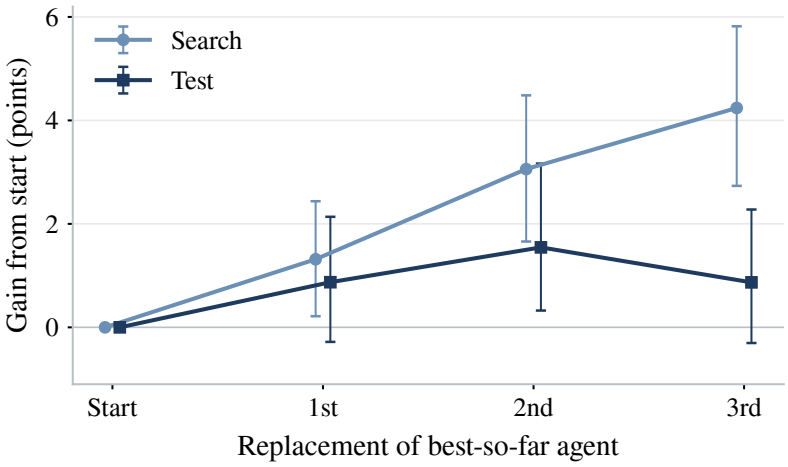

Figure 20: **Recorded search progress outpaces test progress.** Mean gains from the starting agent through three replacements of the best-so-far agent, following the same 33 runs. Error bars are pointwise 95% intervals from resampling runs. Both splits measure the same promoted agents. The gap combines measurement noise and transfer and does not isolate either cause.

Screen size. GEPA's CharXiv training screen uses eight items except at the end of each pass, where four remain. Across 4,808 edits, four-item screens pass 61.7% of children, compared with 53.5% for eight-item screens. Among passed children with search measurements, 41.1% and 37.7% measurably regress their parent, while 4.1% and 4.4% improve. Because screen size follows position within a pass, it is confounded with item composition and potentially candidate history. The comparison motivates controlled changes to screening precision; it does not establish their causal benefit.

Acceptance versus parent effect. Table 39 reports another diagnostic, not a decision-error rate. GEPA's recorded decision is a training screen against the parent; Meta-Harness retains an accuracy–context-length frontier; CRO compares against the incumbent; AdaEvolve's kept edits here are inferred promotions. An edit that improves its parent can fail a different selection criterion, and an edit that regresses its parent can still satisfy one. The expanded cohort contributes 7,853 GEPA, 1,474 Meta-Harness, 7,114 AdaEvolve, and 1,459 CRO measured edits. Columns partition each method's measured edits and sum to approximately 100% across the row; they do not condition separately on kept and rejected edits.

Table 39: **Verdict against effect.** Share of each method's measured edits in the expanded 202-run Luna cohort, by whether the loop kept the edit and whether the edit measurably improved, left unchanged, or regressed the candidate it was built on. AdaEvolve's verdict is inferred from scores.

	kept			rejected		
	improves	noise	regresses	improves	noise	regresses
GEPA	3.2	32.4	21.5	0.2	29.1	13.6
Meta-Harness	4.8	11.5	3.8	5.0	39.8	35.1
AdaEvolve	0.8	4.0	0.2	0.5	13.3	81.2
CRO	8.6	4.4	0.0	6.6	30.9	49.6

Design implication. Selection scores need independent confirmation on the same items to assess execution noise and on fresh items to assess transfer. A stronger gate should be evaluated by the quality of the returned agent, including useful candidates it rejects, rather than by agreement with an uncertain parent-effect label alone.

G.6 Evaluation Overhang

How a stored error persists. Winner's curse concerns which measured score wins. Evaluation overhang concerns what happens when that score remains the threshold for later candidates. Suppose the incumbent's expected score is S but its stored score is $S + e$. A child whose expected improvement is δ receives a fresh error $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. A score-based rule accepts it with probability

$$\Pr[S + \delta + \varepsilon > S + e] = \Phi\left(\frac{\delta - e}{\sigma}\right). \tag{7}$$

Thus a favorable incumbent error can suppress acceptance even when the child is better. More repetitions reduce uncertainty in an initial score, but reusing that score after selection can still carry its favorable error forward. Figure 21 isolates this intervention under the stated model; it is not an estimate of how often it occurs in the main grid.

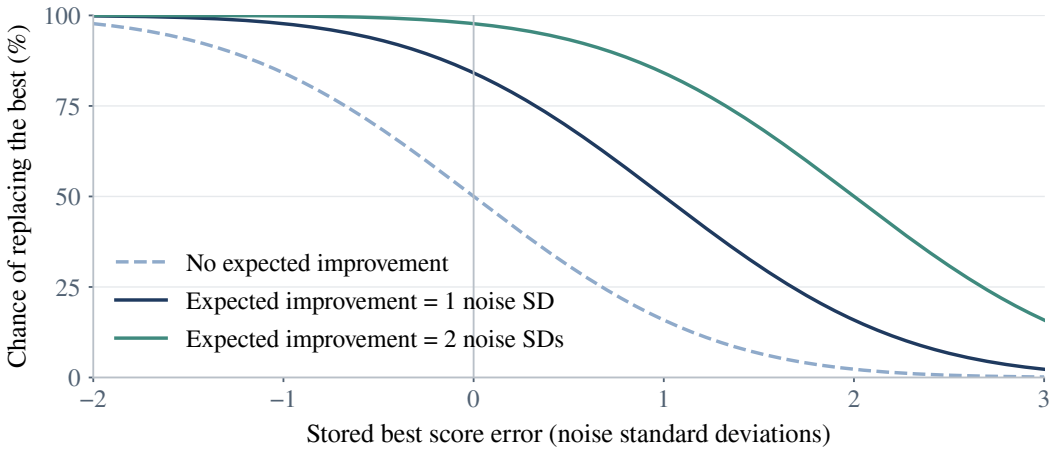

Figure 21: **A favorable stored error raises the next candidate's hurdle.** Illustrative Gaussian calculation with fixed expected child improvements 0, σ, and 2σ. The horizontal axis varies only the incumbent's stored error; the child's evaluation noise stays fixed. Positive error lowers acceptance of both unchanged and genuinely better children.

Limits of the observational check. In the 43-run promotion cohort, the first promotion's search–test step difference correlates negatively with subsequent recorded search gain (-0.26) and final test gain (-0.43). The pattern does not persist consistently at later promotions. The step difference combines noise and transfer, and the first promotion's score also enters the baseline for subsequent search gain. Similarly, correlating the seed reading with gain measured from that reading introduces a shared term: if the predictor is $x - \mathrm{median}(x)$ and reported gain is $y - x$, a negative association can occur without changing the search trajectory. These associations cannot establish a causal overhang effect or justify spending repetitions preferentially at the start.

Design implication. The intervention suggested by the intervention is to refresh a promising candidate and its incumbent on comparable independent evaluations, then update the scores actually used for selection. Merely collecting an extra score without changing the stored threshold does not remove the overhang. A controlled comparison of that policy at fixed total evaluation cost would test whether confirmation offsets the proposals forgone.

H Selection: Further Results

Selection controls both which candidates survive and where subsequent proposals begin. The analyses below distinguish allocating effort to a branch, obtaining another measured improvement on it, and identifying which interventions contribute. None is interchangeable with a confirmed cumulative gain on held-out items.

H.1 Ancestry and Continuation Windows

Main-figure cohort. We use the expanded 202-run Luna annotation subset, retaining 185 runs with at least nine edits. An eligible focal candidate is a retained edit classified as improving its immediate parent under the one-standard-error rule. We reconstruct parent links in recorded edit order and exclude chains with unknown parents, ambiguous repeated content hashes, or self-edges. Normalized edit position preserves chronology when a resumed run resets its step counter. Figure 6 requires twenty subsequent edits with resolved ancestry, giving 219 distinct candidates from 51 GEPA runs, 70 from 28 CRO runs, and 41 from 22 AdaEvolve runs. Candidates discovered too late to supply that window are excluded.

Separating production from retention. A focal candidate receives follow-up once any descendant is edited. It produces a measured improvement once any descendant improves its own parent, whether retained or rejected; the final event additionally requires retaining an improving descendant. Within twenty subsequent edits, these counts are 144, 15, and 14 out of 219 for GEPA; 56, 28, and 19 out of 70 for CRO; and 28, four, and three out of 41 for AdaEvolve. All descendant edits in these eligible windows have measured effect classifications. Intervals resample whole runs (20,000 draws, seed 41), preserving multiple candidates and overlapping windows within a run. Counts pool candidates, so longer runs can contribute more.

GEPA retains another measured improvement on nearly every branch that produces one. This pattern is consistent with its acceptance rule, which compares a child with the parent it edits on the same minibatch rather than requiring it to beat the current best candidate. Its per-example Pareto population can also preserve candidates that are strongest on particular items as possible parents for later search. Our improvement label is parent-relative as well, although it uses the full recorded search evaluation and a one-standard-error threshold, so the comparisons are aligned but not identical. CRO instead requires a candidate to at least match the incumbent and consequently has a larger gap between producing and retaining improvements. A descendant can improve its parent while remaining below that incumbent, so rejection alone does not establish a selection error. These comparisons are observational and do not independently confirm benefit.

Counting branches and edits. The windows contain 552 distinct descendant edits for GEPA, 620 for CRO, and 340 for AdaEvolve. Of these, 19, 61, and five measurably improve their own parent, and 17, 34, and four of those improvements are retained, respectively. One edit can descend from multiple focal candidates, and one branch can contain rejected and retained improvements. Branch counts therefore differ from edit counts; neither provides independent trials for a binomial test.

Follow-up over time. In the same fixed twenty-edit cohorts, GEPA's follow-up counts at five, ten, and twenty edits are 78, 114, 144, with retained measured-improving descendants for 5, 8, 14 candidates, respectively. CRO's follow-up counts at five, ten, and twenty edits are 52, 55, 56, with retained measured-improving descendants for 8, 13, 19 candidates, respectively. AdaEvolve's follow-up counts at five, ten, and twenty edits are 27, 28, 28, with retained measured-improving descendants for 1, 2, 3 candidates, respectively.

Window sensitivity. Table 40 instead admits every candidate eligible for each horizon. Its denominators change across columns and therefore differ from the main figure's fixed-cohort counts. Both constructions show more follow-up than further retained improvement. Equal edit windows do not equalize cost, elapsed time, parent quality, or difficulty. Low proposal yield alone can also make repeated improvement rare; the comparison does not identify a causal selection-policy effect.

Table 40: **Continuation at alternative horizons.** Entries are follow-up / eligible candidates, then further retained improvement / eligible candidates. Cohorts are chosen separately at each horizon.

Method	5 subsequent edits	10 subsequent edits	20 subsequent edits
GEPA	81/238; 5/238	118/232; 8/232	144/219; 14/219
CRO	74/103; 15/103	72/96; 21/96	56/70; 19/70
AdaEvolve	31/46; 1/46	32/45; 2/45	28/41; 3/41

Ancestry is not accumulated performance. Among retained improvements after the first in a run, 73/198 GEPA edits, 45/78 CRO edits, and 10/25 AdaEvolve edits descend from any earlier retained improvement. Looking only at consecutive improvements would miss progress on alternating branches. A descendant can improve its immediate parent after an intervening regression without exceeding the focal ancestor; a branch can also lose interventions or independently reconstruct changes from another branch. These statistics measure structural continuation, not preservation of useful interventions or net held-out gain.

Missing parentage. Meta-Harness has 531 edits with unknown declared parents across 51 eligible runs and remains outside the main comparison. Content similarity does not substitute for known ancestry. AdaEvolve has three ambiguous content hashes, affecting four of 29 later improving edits; 25 cases remain for its ancestry comparison. Its expanded fixed-window cohort now supports inclusion in Figure 6. Excluding unresolved chains restricts the population described; earlier depth summaries based on inferred parents remain outside this comparison.

H.2 Premature Commitment and Parent Choice

What premature commitment means. A high score can attract substantial follow-up before its advantage has been independently checked. The concern is committing resources on insufficient evidence, rather than revisiting a candidate merely because its first measured gain is small. The latter can be reasonable when potential gains are near the evaluation's resolution.

The expanded Luna cohort contains 514 promotions with measured changes against their edited parent: 236 improve, 218 fall within one estimated standard error, and 60 regress (Table 41). Promotion compares candidates under each method's own rule, so a regression against the edited parent need not be a regression against the incumbent. These categories describe the evidence available in the recorded evaluation; none independently confirms the new candidate's advantage.

Table 41: **Initial evidence for promoted edits.** Expanded Luna cohort, all run lengths. Counts compare the promoted edit with its edited parent, not necessarily the replaced incumbent. Unmeasured parent effects are excluded.

Promoted edit versus its parent	Promotions	Share (%)
Measured improvement	236	45.9
Within one standard error	218	42.4
Measured regression	60	11.7

How methods allocate follow-up. GEPA samples parents that perform best on individual items, spreading effort across alternatives. CRO chooses a base per batch and revisits most candidates within five edits; GEPA's follow-up accumulates more gradually. AdaEvolve uses subpopulations and now contributes an expanded resolved-ancestry cohort. Meta-Harness can revisit its history, but incomplete declared parent links still prevent a reliable four-method comparison. A stronger parent may leave less room for improvement, and noisy scores can obscure a useful edit, so observed differences do not isolate the effect of parent choice.

Why parent-policy comparisons are difficult. CRO edits built on the incumbent measurably improve it in 7.0% of cases in the first half of a run and 4.7% in the second; seed-based edits improve in 36.0% and 18.5%, respectively. Of 110 retained measured improvements, 52 start from the seed and 23 from the incumbent. The seed usually has more headroom and can support qualitatively different edits, while incumbent-based edits occur after prior selection. These differences prevent interpreting the rates as the effect of choosing a particular parent. The broader scope of a coding agent also makes declared ancestry essential: a filesystem copy, a retained candidate, and the code most similar to the child need not identify the same source of an edit.

Design implication. Follow-up allocation can use estimated improvement and uncertainty rather than a stored score alone. A controlled test would confirm candidate quality independently before allocating an equal follow-up budget across branches. It should measure final gains over the focal

candidate, alongside the number of useful descendants, to distinguish discovering new gains from recovering intervening losses.

H.3 Coarse Credit Assignment

The attribution problem. Accepting a candidate gives credit to an entire edit, even when that edit changes several interventions. A retained intervention therefore need not have an individually demonstrated benefit. In the first annotation batch, most kept edits add interventions without removing any (69–81% across methods), and at most 3% leave fewer interventions than their parent. This accumulation motivates finer attribution; it does not establish that additions are unnecessary.

Leave-one-out experiment. We examine the returned agents of 16 available CharXiv runs from the minimal seed: six GEPA, six AdaEvolve, three CRO, and one Meta-Harness. GPT-5.6-Sol generates a separate removal variant for each targeted intervention while preserving the others. The saved records contain 119 distinct targets and scored variants. Thirteen removals could not be located or produced no changed lines; six other variants have rollout error rates above 20%. The remaining 100 comparisons comprise 55 GEPA, 30 AdaEvolve, 12 Meta-Harness, and three CRO variants. Each full agent and variant is evaluated on the same 66 test items with three repetitions, using GPT-5.6-Luna. Effects compare item-paired scores; positive values mean removal lowers performance.

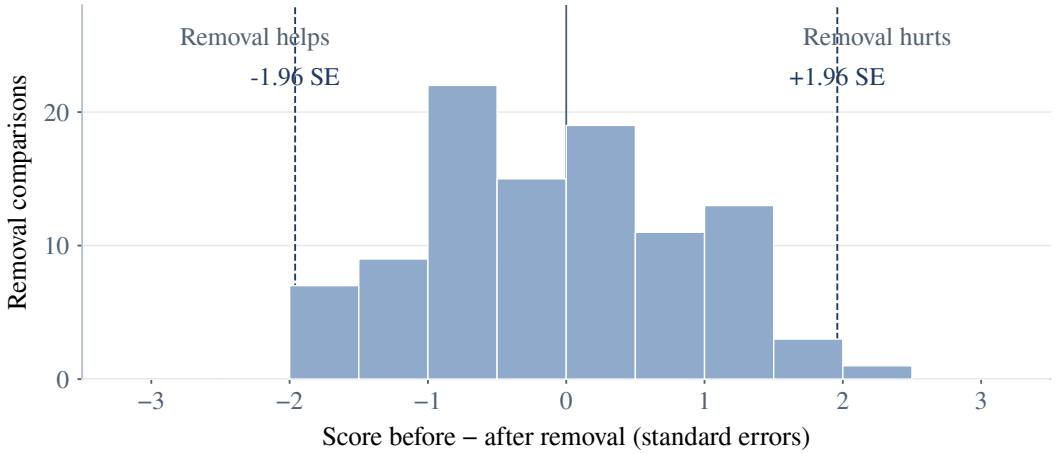

Figure 22: **Intervention-removal effects have limited resolution.** Full-agent minus ablated-agent score, divided by its estimated standard error, for 100 usable comparisons from 16 agents. Dashed lines mark ± 1.96 standard errors, crossed once in each direction. Comparisons from one agent share a full-agent baseline; the histogram does not treat them as independent tests.

What the ablations establish. At one standard error, removal lowers the score in 17% of comparisons, changes it by less than that threshold in 66%, and raises it in 17%. At 1.96 standard errors there is one crossing in each direction. The measurements therefore provide little precise attribution at this evaluation budget. Nondetection is not evidence that an intervention is inert: small effects, redundancy, and interactions can all produce weak individual-removal effects. Generated deletions may also modify more or less than the intended intervention. Invalid removals do not establish architectural indispensability, and individually small effects do not imply that joint pruning is safe.

Design implication. Useful credit assignment requires interventions whose scope is checked, sufficient precision to resolve effects of interest, and tests of interactions before combining or removing interventions. The present experiment motivates that measurement problem; it does not validate a pruning rule or show that most accumulated changes can be discarded.

I Research Directions by Failure Mode

This roadmap expands Section 7 and follows the nine failure modes in Table 19. The analyses motivate these directions but do not establish that the proposed changes improve optimization. Comparisons should account for both optimizer and worker costs and assess returned agents on examples withheld from search. Each item below connects a finding to a concrete design response and an outcome by which to judge it.

I.1 Proposal: Generate and Test Better Ideas

1. **Mode collapse: broaden the interventions considered and connect them to observed failures.** Concentrated proposals and low measured yield motivate training proposers to generate distinct, task-specific interventions, rather than superficial variations of familiar instructions. A method could ask for competing explanations of an observed error, gather evidence that distinguishes them, and implement a change with a stated prediction about which failures it should repair. The loop should permit investigation without requiring every step to produce an edit. The controlled feedback probe shows that examples can redirect proposals; it does not establish that greater diversity improves performance. Success should therefore require independently verified gains from the proposed changes, alongside intervention-level diversity and checks that the targeted failures are repaired (Appendices F.1 and F.5).

2. **Retrying rejected ideas: retain the evidence behind a rejection.** A useful search history should record the intervention, its implementation, the parent agent, the observed outcome, and the uncertainty of that outcome. Before retrying an idea, the proposer could identify what has changed: a different implementation, a different parent, new failure evidence, or an earlier evaluation too imprecise to resolve the effect. This would support targeted retries instead of either forgetting past attempts or permanently banning rejected ideas. The recorded-sequence analysis shows that blanket retry suppression would also remove measured improvements. A revised policy should be judged by useful gains preserved and total cost, not merely by the number of repeats eliminated (Appendix F.6).

3. **Rapidly diminishing returns: adapt effort to the prospect of further gains.** Optimizers could estimate the chance and value of another confirmed improvement from recent evaluations, their uncertainty, and the quality of the current parent. Falling yield could trigger a change in the kind of proposal, a return to an earlier parent, a shift to another branch, or stopping. Such decisions must use information available at the time; the last promotion and normalized final run position are retrospective quantities. A useful policy should obtain comparable or better held-out performance at lower total cost, or better performance at the same cost, while accounting for beneficial late discoveries it might miss. The observed decline alone does not establish which response is best (Appendix F.7).

I.2 Evaluation: Separate Apparent Progress from Confirmed Gains

4. **Winner's curse: confirm apparent winners with fresh measurements.** A high recorded score could trigger a new comparison of the candidate and incumbent on the same items, with additional repetitions where uncertainty could change the decision. The confirmation must use executions collected after candidate selection, rather than reuse the measurements that made the candidate look best. Measurement rules should account for repeated decisions if they claim a particular error rate. A final evaluation unused in candidate choice should support the reported gain. The relevant tradeoff is whether more reliable choices offset the proposals forgone at a fixed total budget; lower measurement variance alone is not sufficient evidence of a better optimizer (Appendices G.3 and G.2).

5. **Evaluation overhang: refresh the score that later candidates must beat.** An additional evaluation changes search only if its result updates the stored comparison score or the acceptance rule. A method could replace a single lucky incumbent reading with an estimate based on fresh repetitions, tracking its uncertainty as later candidates are compared. Starting agents and incumbents should be evaluated under comparable conditions. This direction addresses the persistence of an inflated threshold, whereas the preceding item addresses how a favorable measurement wins

selection in the first place. Assessment should track useful candidates admitted or rejected, the quality of the returned agent, and the cost of refreshing scores (Appendix G.6).

6. **Overtrusting dev scores: check transfer separately from repeatability.** Repeated executions on the same examples can establish a repeatable search gain without showing that it generalizes. Optimizers could use additional training-side validation batches to check whether proposed changes depend on particular search examples, while preserving a final test set that never supplies optimization feedback. Pairing candidate and baseline measurements helps assess score differences, but cannot by itself correct adaptation to the selected examples. A successful design should improve the returned agent on withheld examples, not simply narrow uncertainty around its search score. The independently reevaluated CharXiv results illustrate why both checks are needed (Appendices G.5 and G.2).

I.3 Selection: Develop and Preserve Useful Discoveries

7. **Failure to compound gains: match the response to where progress stops.** Methods should distinguish whether an improved candidate receives follow-up, whether that work produces another improvement, and whether an improvement is retained. Where follow-up rarely produces gains, possible responses include changing how proposals develop the current agent or which parents receive effort. Where improving descendants are rejected, the candidate should be compared with the incumbent, since beating its own parent is insufficient. Parent-policy comparisons should account for headroom and follow-up cost. To establish compounding, measure independently confirmed net gain over the original improved candidate: recovering an intervening loss or merely extending its ancestry is not enough (Appendices H.1 and H.2).

8. **Premature commitment: make follow-up investment depend on the strength of the evidence.** Before devoting substantial work to an apparent winner, an optimizer could check whether its advantage survives fresh evaluation. Follow-up allocation could then consider the estimated gain, its uncertainty, and the opportunity to improve other candidates. Uncertainty should not automatically disqualify a candidate: an unresolved but promising alternative may justify a small investigation budget. The distinction is between testing its potential and committing most of the search to it as an established improvement. Success should be assessed through the final independently measured gain and the useful alternatives that the policy overlooks, rather than through fewer visits to uncertain candidates alone (Appendix H.2).

9. **Coarse credit assignment: identify which interventions to preserve and combine.** Methods could keep separable records of individual interventions and use targeted additions or removals to investigate their contributions. Such comparisons require checking that the intended intervention was actually changed, collecting enough repeated evaluations to resolve relevant effects, and accounting for shared baselines. Interactions also matter: a change with little effect in isolation may be useful in combination, and several individually harmless removals may jointly damage performance. The present removal experiment has limited resolution, so it does not justify deleting all changes with undetected effects. The objective is to preserve independently supported gains while making further edits, not simply to reduce the number of interventions (Appendix H.3).

J Validation of Search Dynamics: Full Model and Analysis

J.1 From Testbed Observations to a Model

Why a smaller testbed? To distinguish program quality from score noise, we must execute unchanged programs repeatedly on the same questions. Doing this for many programs with frontier models is expensive. We study GEPA editing a question-answering program built around Qwen3 on HotpotQA, where repeated evaluations are more affordable. Holding both program and questions fixed reveals variation due to execution; comparing independently reevaluated programs reveals differences in expected score (Appendix K).

In the earlier Qwen3-8B measurements, 24 edits of the starting program have a mean gain of 3.2 percentage points and an estimated standard deviation of 2.6 points after accounting for evaluation noise. A Normal distribution provides a simple approximation, although the small sample and vis-

ible skew do not establish that shape. Measurements at higher parent accuracies suggest smaller mean gains and less variation between edits (Appendix Figure 25). These observations suggest that improvement becomes harder as the parent improves. They inform the model's form; its parameters are fitted separately in each prediction setting.

A model of proposed edits. In the detailed notation below, J and Δ correspond to the expected search score S and single-edit effect δ in Eq. 1; Δ here is not the run-level gain Δ_{search}. Let J be the expected search accuracy of the program being edited (the parent), and Δ the change caused by an edit. A setting consists of a base model, task, optimizer, method, and serving route; different starting agents share its fit. For setting c, we model

$$\Delta \mid J, c \sim \mathcal{N}\big(\mu_{c,0} - b_c(J - J_{c,0}), s_c^2\big)\,. \tag{8}$$

Here $\mu_{c,0}$ is the mean gain at reference accuracy $J_{c,0}$, b_c describes how that mean changes with accuracy, and s_c is the standard deviation of edit effects. We initially hold s_c constant. The testbed suggests that this simplification may fail as accuracy rises, a possibility examined below.

Fitting from two parents. The starting program's repeated outcomes estimate the noise in one search score. Scores of multiple edits to the same parent then contain two sources of variation: differences between the edited programs and evaluation noise. We estimate the former while accounting for the latter, rather than treating every observed difference as a real effect (Appendix J.4).

We reserve one run per setting and select two parents at different accuracies, each with at least three evaluated children. Their mean edit gains determine the line in Eq. 8; the child scores also estimate the shared standard deviation. We carry uncertainty in these fitted quantities into predictions. Parent selection follows a fixed ordering and measurement-availability rule, without using prediction accuracy. The reserved run is excluded from prediction results. Using the first two parent stages would make calibration possible early in search; the available main-grid measurements can come from later stages, so our study does not yet demonstrate calibration before optimization starts (Appendix J).

J.2 How Proposal Effects and Noise Shape Progress

Starting from a target run's initial score and estimated noise, the simulator proposes an edit to its current best program, adds evaluation noise, and retains the child if its measured score exceeds the stored best. It tracks two curves: the *reported gain* used for selection, and the selected program's *underlying gain* on the same search questions. The latter removes measurement error from the score, while selection still uses noisy evaluations. Their difference shows how much apparent progress comes from favorable measurement errors relative to the initial reading. Neither curve predicts performance on new questions.

Figure 23 shows why both curves matter. Even a proposer whose edits are harmful on average can accumulate improvements when some edits help and evaluation distinguishes them. Greater variation between proposals creates opportunities for larger gains. Higher evaluation noise can instead produce a rising reported score while the selected program stagnates or deteriorates. As mean proposal gain declines with parent accuracy, progress slows. The resulting flattening depends on the proposer, noise, and evaluation budget; it is not an intrinsic ceiling of the task.

J.3 Validation on Runs Excluded from Fitting

Having fitted a setting once, we forecast each other run using only its initial evaluation, without fitting to its later outcomes. We compare simulated and recorded gains at the same number of evaluations; predicting the stopping time is a separate problem. The main comparison uses Meta-Harness, whose runs we approximate with the incumbent-editing simulator. Other methods' endpoint comparisons are reported as stress tests because their parent and selection rules differ (Appendix J).

Measurement budget. For Luna/Opus/GPQA, fitting uses eight full-search evaluations. Forecasting eight other runs adds one initial evaluation each: **16 measurements versus 250 across the completed target runs (6.4%)**. Across all eight settings, this fraction ranges from 5.4% to 41.1% (Table 42). These counts quantify the data used for prediction. They exclude the search needed to produce the archived calibration programs and therefore do not establish end-to-end dollar savings.

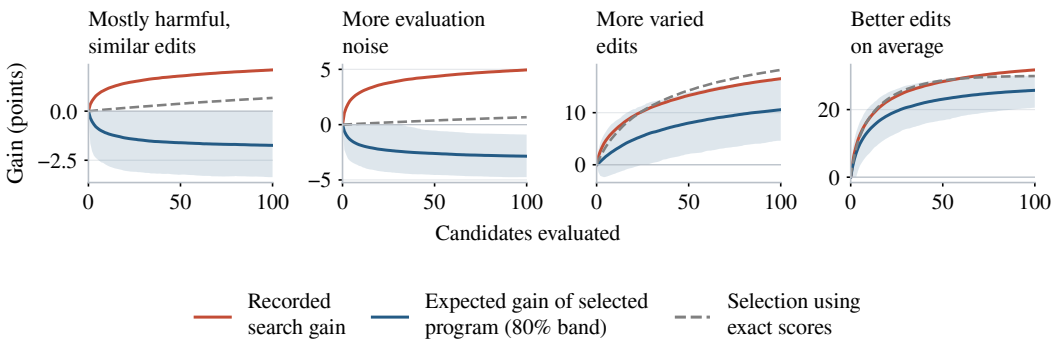

Figure 23: **How proposer behavior and evaluation noise shape a run.** Illustrative simulations starting at 70% accuracy, with mean proposal gain $\mu_0 - 0.3(J - 70)$, proposal spread s, and evaluation noise σ in percentage points. Reading left to right, (μ_0, s, σ) is $(-2, 1, 1.7)$, $(-2, 1, 3.2)$, $(-2, 4, 3.2)$, and $(3, 4, 3.2)$; one parameter changes each time. Red is mean recorded gain. Blue is mean expected search gain of the noisily selected program, with the middle 80% of runs. Grey uses exact scores for selection. Each setting has 4,000 runs of 100 candidates; vertical scales differ.

Prediction results. Across 38 Meta-Harness runs excluded from fitting, mean absolute error in reported gain is 2.51 points, with 29/38 (76.3%) inside the model's conditional 80% intervals. Figure 24 shows three successful settings selected after inspecting results: one trajectory and the final gains of all tested runs in two other settings. The trajectory compares predicted and observed progress throughout a run; the endpoint panels show whether the same fit works across starting agents and random seeds. Appendix Table 43 reports all eight settings, including failures.

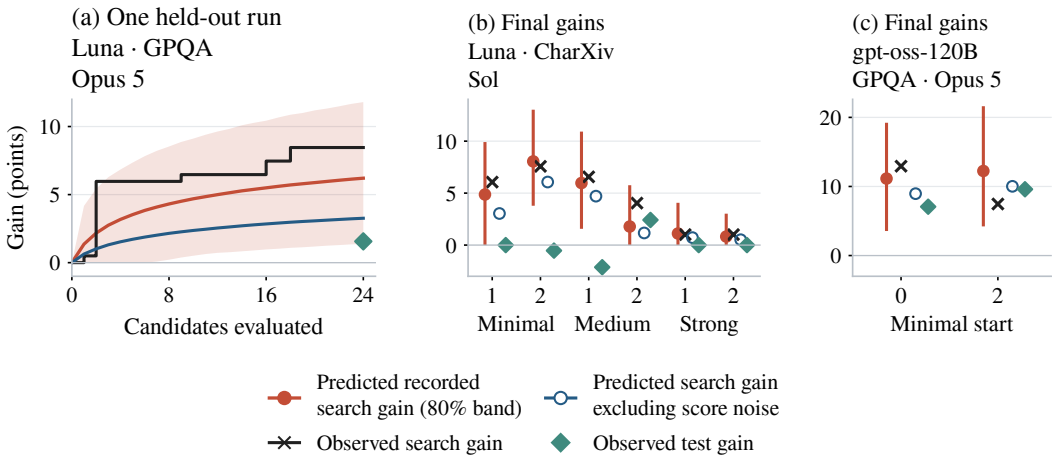

Figure 24: **Three selected successful prediction settings.** (a) A held-out run chosen by fixed ordering. (b,c) Final gains of all tested runs in two other settings; numbers identify runs within starting tiers. Red predicts the score recorded during search, with conditional 80% bands or intervals; black is the observed search gain. Blue predicts expected gain on the same search items with score noise removed, while selection remains noisy; it is not a test-gain forecast. Green is observed test gain. Axes use percentage points and different vertical ranges. Settings were selected after inspecting results; all-setting errors, coverage, and failures are in Table 43.

A separate check using the first two parents. On Qwen3-14B/HotpotQA, we fit edits of the seed and its first accepted child from one run, then predict mean edit gains in 12 parent groups from two other runs. The error is 1.29 points, versus 5.11 for a constant-gain baseline. However, unchanged parent scores shift under the reconstructed serving environment; a parent-shift sensitivity gives errors of 2.15 versus 4.44 points. Both fits leave the decline in mean gain uncertain. This exploratory result

checks edit effects at measured parent accuracies, not full-run forecasts or GEPA's selection policy (Appendix J.6).

What the failures reveal. On Luna/Opus/CharXiv, predicted gain is 27.63 points against 12.37 observed, with neither run inside its interval. The higher-scoring calibration parent's children have less variable scores. Fitting one standard deviation can keep large improvements too likely late in a simulation; allowing it to change with parent accuracy helped in an exploratory follow-up. Coverage also falls to 4/8 on Luna/Sol/GPQA and 0/2 on Haiku/Opus 4.8 (Appendix J.5).

Selecting a parent for a high measured score can bias the fitted edit gains. Our intervals include uncertainty in proposal means and variation, but omit uncertainty in parent scores and noise estimates, and assume independent edits. The simulator always edits the incumbent, whereas Meta-Harness can revisit earlier parents and other methods use populations or additional selection rules. Finally, underlying search gain need not transfer to test: the independent CharXiv reevaluations confirm search gains without comparable test gains (Appendix Figure 17). Agreement with search progress supports the account of observed dynamics, but does not establish generalization.

J.4 Noise, simulation, and illustrative regimes

Fitting from two parents. The seed's first evaluation estimates the noise in one reported score. With N questions and $K = 3$ independent repetitions per question,

$$\hat{S} = S + \varepsilon, \quad \varepsilon \approx \mathcal{N}(0, \sigma^2), \qquad \widehat{\sigma}^2 = \frac{1}{N^2} \sum_{i=1}^{N} \frac{\widehat{v}_i}{K}, \tag{9}$$

where S is the expected score and $\widehat{v}_i$ is the sample variance of repeated outcomes on question i. We use this noise estimate for edited programs, with a check using each edited program's own repetitions.

We reserve one run for fitting and choose two parents at different accuracies, each with at least three evaluated edits. Within each group, observed child-score variance combines proposal variation and evaluation noise: $s_c^2 + \widehat{\sigma}^2$. We estimate the two mean gains and shared spread together, and carry their uncertainty into the predictions. The two means determine the line $\mu_c(J)$. Parents are chosen by a fixed ordering and measurement-availability rule, not by prediction accuracy; they may occur anywhere in the reserved run (Appendix J).

Predicting another run. For a run excluded from fitting, we use its initial search score and noise estimate. The simulated loop proposes an edit to its current best program, evaluates the result, and keeps it if its score exceeds the stored best score. After t candidates, its reported gain and underlying gain on the same search questions are

$$\hat{C}_t = \max_{0 \le u \le t} (S_u + \varepsilon_u) - (S_0 + \varepsilon_0), \tag{10}$$

$$C_t = S_{u^\star} - S_0, \qquad \hat{C}_t - C_t = \varepsilon_{u^\star} - \varepsilon_0, \quad u^\star = \arg\max_{0 \le u \le t} (S_u + \varepsilon_u). \tag{11}$$

Here $u = 0$ denotes the seed and $u^\star$ the selected candidate. The underlying gain C_t removes measurement error from its score, while selection still uses noisy evaluations. We compare predictions at each run's recorded number of evaluations; predicting when the method will stop is a separate problem.

Illustrative regimes. Figure 23 varies proposal means, proposal variation, and evaluation noise to explain their effects on reported and underlying search gains. These simulations illustrate the model's behavior; the held-out prediction results are reported separately in Figure 7 and Table 43.

J.5 Calibration Cohorts and Measurement Budget

Scope and data. The updated analysis audits 244 archived run records from the original and late main grids, the available weaker-base side grids, and three completed remote runs missing locally. Of these, 198 pass run-level availability and validity checks. Twenty-three setting/worker-route

groups admit a two-parent fit, yielding 109 held-out endpoint comparisons; 12 other eligible groups lack sufficient full-search-set parent/child measurements. Forty-six records fail run-level checks, including explicit invalidity flags, seed-only results, missing original seeds, incomplete runs, or absent usable evaluation/parent records. No new optimization runs or evaluation calls are used for these grid comparisons. The completed Qwen top-up is a separate testbed check described below.

The common incumbent simulator is evaluated on 38 held-out Meta-Harness runs, which form Table 43. Other-method endpoints are stress tests of the same approximation, not native-method predictive validation. In particular, AdaEvolve's training-set evaluations do not generally provide the full-search-set child groups needed by this fit. Historical Haiku traces identify Claude Opus 4.8, so these are not pooled with Opus 5. Worker serving routes are also separated. Some late Databricks runs have anomalously low initial readings and large apparent gains; their route-specific diagnostics are not used as evidence for the OpenAI-route results in the table.

Calibration selection. A class fixes base model, task, optimizer model, method, and worker route. Runs are ordered by random seed, then starting tier (minimal, medium, strong). We reserve the first run with a qualifying parent pair. Parents are ordered by the first proposal with a measured child. Each must have at least three distinct children scored on exactly the same full search items; their observed score gap must be at least 5 points and 1.96 estimated measurement standard errors. If no run admits that separation, the earliest distinct-score pair with the required child counts is retained as an explicitly exploratory fit. The calibration run is excluded from prediction metrics. No pair is chosen by held-out prediction accuracy. This availability-based choice and the overall analysis are exploratory, not preregistered; the displayed successful settings were chosen after inspecting the results.

Evaluation budget. We count full-search-set score evaluations, including the initial evaluation in each complete run. The fitting count F is the union of the seed, two parents, and their measured children, counting a program once even when it serves multiple roles. Each additional forecast uses one initial evaluation. For n target runs with H evaluations in total, the measurement fraction is $(F + n)/H$. The reserved run's complete evaluation count R is also shown: it records the search from which fitting snapshots were obtained and is not included in F. Thus the table measures data requirements, not demonstrated end-to-end savings. It excludes proposal generation, partial evaluations, test evaluation, and simulation compute. Fitting scores use three repetitions per item; some target scores use other repetition counts, and token use also varies, so these ratios are not rollout or dollar-cost ratios.

Table 42: **Measurements used for forecasting versus complete target runs.** All eight Meta-Harness settings, including prediction failures. F: distinct programs evaluated for fitting; R: evaluations in the complete reserved run; n: target runs; H: their total full-search-set evaluations. The last column amortizes fitting across targets and includes each target's initial evaluation.

Base	Task	Optimizer	F	R	n	H	$(F+n)/H$
Luna	GPQA	Opus 5	8	35	8	250	6.4%
Luna	GPQA	Sol	11	25	8	223	8.5%
Luna	CharXiv	Opus 5	56	109	2	146	39.7%
Luna	CharXiv	Sol	19	43	6	465	5.4%
Luna	τ^2	Opus 5	18	24	8	282	9.2%
Luna	τ^2	Sol	28	30	2	73	41.1%
gpt-oss-120B	GPQA	Opus 5	12	31	2	60	23.3%
Haiku 4.5	GPQA	Opus 4.8	25	33	2	68	39.7%

Likelihood and uncertainty. Let y_{gj} be child j's observed score for parent group $g \in \{0, 1\}$. Conditional on observed parent scores, the likelihood is

$$y_{gj} \mid m_g, s \sim \mathcal{N}(m_g, s^2 + v_{gj}).$$

The primary fit takes every v_{gj} to be the calibration seed's estimated evaluation variance (Eq. 9); a sensitivity uses each child's own estimate. Flat priors on m_0, m_1 are integrated analytically. The spread has a half-Normal prior of scale 5 percentage points and is integrated on a grid from 0 to 40

points at spacing 0.005. Parent means give $\mu_g = m_g - \hat{J}_g$ and $b = (\mu_0 - \mu_1)/(\hat{J}_1 - \hat{J}_0)$; b is not constrained to be positive. Uncertainty in m_g and s propagates to the predictions, conditional on the parent scores and noise estimates. There is no correction here for selection of the parents, nor an assumption that a posterior median above zero establishes nonzero intrinsic variation.

For example, Luna/Sol/GPQA has three and six measured children with observed score standard deviations 1.25 and 0.98 points, compared with seed evaluation noise of 1.72 points. Direct subtraction gives negative intrinsic-variance estimates. The shared-spread posterior instead has median 0.61 points and a 90% interval of $[0.06, 2.02]$. Using child-specific noise gives a median of 0.56. In this setting, changing the half-Normal scale to 2.5 or 10 points changes the median to 0.57 or 0.62; this sensitivity does not establish robustness in every setting.

Simulation and outcomes. We simulate 12,000 trajectories for each target and retain one parameter draw per trajectory. A target's latent starting accuracy is drawn from a Normal centered on its initial reading with its estimated evaluation variance, clipped to $[0, 100]$; the stored initial reading remains the observed value. Each proposed latent score is clipped to $[0, 100]$, then receives independent Gaussian evaluation noise. The candidate replaces the incumbent only if that reading is larger. The number of full-search-set evaluations minus the initial evaluation supplies the comparison horizon; this is not a prediction of the stopping time. Repeated evaluations, proposal dependence, and historical-parent choices are not modeled explicitly. Intervals are conditional on this approximation and omit uncertainty in calibration-parent accuracy and estimated noise.

Observed search gain is the largest search reading minus the original seed's reading. Test gain uses the existing corrected standardized original-seed baselines for Luna and measured original-seed baselines for the weaker models. Resumed-run corrections prevent a resumed incumbent from being treated as the original seed. Test outcomes are never fitted. The predicted latent score is an expectation on the fixed search items, not on the test population, so test gain is a separate descriptive comparison.

Table 43: **All held-out Meta-Harness settings under two-parent calibration.** Means and mean absolute errors (MAE) are in percentage points. Predicted observed gain includes evaluation noise; predicted latent gain is the underlying search gain of the noisily selected candidate. Coverage is the share of observed search gains inside the conditional 80% interval. Asterisks mark closer-parent exploratory fits. Starting tiers and random seeds are pooled only within each row.

Base	Task	Optimizer	n	Pred. obs.	Pred. latent	Search	Test	MAE	Coverage
Luna	GPQA	Opus 5	8	4.35	1.84	5.16	0.39	1.30	8/8
Luna	GPQA	Sol	8	4.14	1.69	3.86	0.45	1.57	4/8
Luna	CharXiv	Opus 5	2	27.63	24.04	12.37	4.52	15.25	0/2
Luna	CharXiv	Sol	6	3.77	2.70	4.38	−0.04	0.80	6/6
Luna	τ^2	Opus 5*	8	9.42	3.77	7.20	2.08	2.45	7/8
Luna	τ^2	Sol	2	11.92	7.56	9.70	−0.39	2.22	2/2
gpt-oss-120B	GPQA	Opus 5	2	11.69	9.49	10.20	8.33	3.29	2/2
Haiku 4.5	GPQA	Opus 4.8*	2	6.65	4.47	3.48	0.51	3.17	0/2

Interpreting successes and failures. Across all 38 runs, reported-gain MAE is 2.51 points and coverage is 29/38. Figure 24 shows three selected successful settings: one fixed-order trajectory from Luna/Opus/GPQA and all held-out endpoints in the other two settings. The table includes every tested setting, including the unfavorable ones. With only two runs in several rows, neither full coverage nor zero coverage gives a precise population coverage estimate. On Luna/Opus/CharXiv, a shared intrinsic spread of about 4 points combines calibration groups with raw spreads 5.31 and 2.32 points, and overpredicts later progress. An earlier exploratory variable-spread sensitivity predicted 15.17 and 14.10 points for the two targets, against 13.13 and 11.62 observed, but that alternative was examined after inspecting failures and is not the fitted model reported in this table. The Haiku fit uses less-separated parents and framework-inferred ancestry, both further limitations.

J.6 Separate Qwen Two-Parent Check

We reserve Qwen3-14B/HotpotQA/GEPA seed 10 for fitting and seeds 11 and 12 for validation. The two parents are the starting program and its first accepted child. The fitting sample includes all 10

and 6 proposals from these parents within the first 30 iterations; it is not restricted to the first two consecutive iterations. Six previously unmeasured rejected children and both unchanged parents were evaluated on the same 300 population-subset items with two fresh repetitions each: 4,800 valid rollouts, with no missing scores. The other ten child measurements were already archived. The original seed has four repetitions per item and the other archived fitting scores have two. These measurements are independent of the scores used to select candidates and are on a different item set from the original search scores.

The original serving environment could not be recovered exactly. Fresh parent scores are 23.50% and 40.33%, compared with archived scores of 21.25% and 43.50%. The changes are $+2.25$ and -3.17 points, with approximate fixed-item measurement standard errors of 1.08 and 1.32 points. We therefore report the combined fit as a sensitivity analysis, without assuming that the two serving environments are interchangeable. Its likelihood uses the same shared-spread model and half-Normal prior of scale 5 as the main-grid fit, with the archived seed's noise estimate. The adjustment in Table 44 subtracts each parent's observed shift from its three newly measured children while retaining the archived parent scores. This assumes that a parent's shift applies equally to its children; it is not a validated correction. Intervals condition on the parent scores, noise estimate, and, where used, adjustment.

Table 44: **Qwen two-parent fitting sensitivities.** Mean gains μ_0, μ_1 and median proposal spread s are in percentage points; b is the decline per point of parent accuracy, with its conditional 95% interval. MAE compares predicted and measured mean edit gains, averaging equally over 12 held-out parent groups. The constant baseline uses the corresponding fitted seed-stage mean.

Fit	μ_0	μ_1	s	b [95% interval]	MAE	Constant MAE
Combined, unadjusted	5.99	1.96	9.64	0.18 [-0.27, 0.64]	1.29	5.11
Parent-shift adjustment	5.32	3.54	9.96	0.08 [-0.39, 0.55]	2.15	4.44

All fitting choices are held fixed across the two sensitivities. Validation uses 50 measured children in 12 eligible parent groups from seeds 11 and 12; gate-pass and gate-fail means are weighted by each group's recorded proposal frequencies because archived rejected children were subsampled. Nine groups lie outside the fitting parents' accuracy range. Parent groups share runs and sometimes programs, so they are not independent replicates. Their measured parent accuracies enter only as validation covariates. Both fits improve on the constant baseline, but neither establishes a positive decline, a Gaussian proposal distribution, or an accurate simulation of GEPA's population selection. Raising the spread-prior scale to 10 gives unadjusted $s = 10.58$ and $b = 0.18$ with interval $[-0.33, 0.69]$; using child-specific measurement noise gives $s = 9.62$ and $b = 0.18$. These checks do not resolve the serving discrepancy.

J.7 Evaluation Noise in the Main and Side Grids

Table 45 summarizes repeated-evaluation noise on fixed search items. These are measurement-noise estimates, not uncertainty in a method's mean gain or a measure of transfer to new examples. The two-parent forecasts use their calibration and target noise estimates rather than pooled proposer fits.

Table 45: **Evaluation noise**: standard deviation of one search-set reading in points ($K = 3$ repetitions per item), $\sigma^2 = \bar{w}/(NK)$ with $\bar{w}$ estimated from each item's repetitions (bias-corrected). Median over all full search-set evaluations, and over the seed agent's first evaluation for each seed agent.

base model	task	N	all evaluations	seed (minimal / medium / strong)	evaluations
GPT-5.6-Luna	GPQA-Diamond	67	1.72	1.76 / 1.76 / 1.22	2,370
GPT-5.6-Luna	CharXiv	66	1.68	1.60 / 1.56 / 1.75	2,942
GPT-5.6-Luna	τ^2-bench	55	3.15	3.26 / 3.15 / 3.15	719
Claude Haiku 4.5	GPQA-Diamond	67	2.22	–	324
gpt-oss-120B	GPQA-Diamond	67	1.86	–	425

K Selection-Noise Derivations and Testbed Measurements

Scope and notation. This appendix gives a fixed-item noise model and measurements from GEPA on HotpotQA with Qwen3 models. Expected accuracy is a population quantity over executions; repeated evaluations estimate it with residual uncertainty. The validation set V here is the main paper's search set D_{search}, R denotes repetitions per item (K in the main text), and $J_V(c)$ and $\hat{J}_V(c)$ denote expected and measured scores. The linear proposal mean below has slope $-(a+b)$, corresponding to the accuracy dependence in Eq. 8; its bounded fix/break parameterization is separate from the two-parent fit. All selection formulas state their assumptions. The incumbent-only rule does not reproduce GEPA's screens and Pareto parents, AdaEvolve's training selection, or CRO's two-stage gate.

Discrete optimizers such as GEPA (Agrawal et al., 2025), AFlow (Zhang et al., 2025b), and AdaEvolve (Liu et al., 2026) keep a pool of candidates and choose among them by their scores on a validation set: which candidate to keep, which to build on, and, at the end, which one to return. With stochastic decoding, every validation score is itself a random variable. This appendix models that selection noise to answer two questions. How large is it relative to the progress the optimizer is making? And how many reruns of the validation evaluation does it take to counteract it?

K.1 Setup

The validation set $V = \{x_1, \ldots, x_N\}$ is fixed, and every selection decision scores candidates on all of V. A candidate c (a prompt, a workflow, a program) solves item x_i with probability $p_c(x_i)$, its *solve rate*, because decoding is stochastic. Its true validation score is

$$J_V(c) \;=\; \frac{1}{N}\sum_{i=1}^{N} p_c(x_i). \tag{12}$$

The optimizer instead observes the average of R independent runs per item ($R > 1$ means reruns):

$$\hat{J}_V(c) \;=\; \frac{1}{NR}\sum_{i=1}^{N}\sum_{r=1}^{R} y_{c,i,r}, \qquad y_{c,i,r} \sim \text{Bernoulli}\big(p_c(x_i)\big). \tag{13}$$

Optimizers typically use $R = 1$. Several also cache model outputs, as DSPy (Khattab et al., 2024) does by default, so that single draw is frozen and reused in every later comparison. Selection noise is the gap $\hat{J}_V(c) - J_V(c)$. Because V is fixed, it comes entirely from the runs. The separate question of whether a gain on V carries over to test data is one of distribution shift, including the overfitting that repeated reuse of a fixed validation set can cause (Dwork et al., 2015; Recht et al., 2019). We do not model it here.

K.2 Noise in one validation score

Proposition 1 (Validation-score noise)**.**

$$\mathbb{E}\big[\hat{J}_V(c)\big] = J_V(c), \qquad \text{Var}\big[\hat{J}_V(c)\big] \;=\; \frac{\bar{w}_V(c)}{NR}, \qquad \bar{w}_V(c) = \frac{1}{N}\sum_{i=1}^{N} p_c(x_i)\big(1-p_c(x_i)\big). \tag{14}$$

This is the within-item (resampling) component of the variance decompositions used for LLM evaluation (Miller, 2024; Madaan et al., 2024), specialized to a validation set that is fixed and reused for selection. The standard error falls as $1/\sqrt{R}$: halving it takes four times as many reruns, and each rerun costs another full pass over V for every candidate. The noise is carried only by items the candidate sometimes solves. An item it always solves or always fails has $p(1-p) = 0$ and contributes nothing.

On HotpotQA with Qwen3-8B ($N = 300$), $\bar{w}_V \approx 0.051$, so a single validation score has a standard error of $\sqrt{0.051/300} \approx 0.013$, that is 1.3 points at $R = 1$ (accuracies, gaps and their standard deviations are quoted in percentage points, $\times 100$, throughout). As an illustration, the same seed program on the same validation set scored 41.0 in one GEPA run and 38.0 in another, one draw

each. This is a 3-point swing with no change in the program. Across programs on one task, $\bar{w}_V$ tracks the accuracy: we write $\bar{w}_V \approx \rho\, J_V(1 - J_V)$ and call ρ the *rollout noise* of the base model on the task ($\rho = 1$ is a program that solves every item with the same probability; $\rho = 0$ solves each item deterministically). For Qwen3-8B on HotpotQA, $\rho = 0.214$.

K.3 Comparing two candidates

Let c and c' be two candidates, for instance a new proposal and the incumbent. Their true gap on V is $\Delta_V = J_V(c) - J_V(c')$, and its estimate is $\hat{\Delta}_V = \hat{J}_V(c) - \hat{J}_V(c')$, from independent runs on the same items.

Proposition 2 (Paired validation noise)**.**

$$\mathrm{Var}\big[\hat{\Delta}_V\big] \;=\; \frac{\bar{w}_\Delta}{NR}, \qquad \bar{w}_\Delta = \bar{w}_V(c) + \bar{w}_V(c'). \tag{15}$$

Under a normal approximation (Φ and ϕ are the standard Normal distribution and density), a decision that follows the sign of $\hat{\Delta}_V$ is correct with probability $\Phi\big(|\Delta_V|\sqrt{NR/\bar{w}_\Delta}\big)$. Requiring this to be at least $1-\alpha$ (one-sided, with $z_q = \Phi^{-1}(1-q)$ and $z = z_\alpha$) gives the runs per item needed, a standard sample-size calculation (Cohen, 1988).

Proposition 3 (Runs per item for a pairwise decision)**.**

$$R^\star \;=\; \frac{z^2\, \bar{w}_\Delta}{N\, \Delta_V^2}. \tag{16}$$

Because $R^\star \propto 1/\Delta_V^2$, halving the gap the optimizer needs to detect multiplies the reruns by four. Because $R^\star \propto 1/N$, a benchmark with a larger validation set needs proportionally fewer reruns for the same gap. The rollout noise $\bar{w}_\Delta$ is a property of the base model on the task, namely how much probability mass sits on items it sometimes solves. Proposition 2 assumes both scores are drawn fresh. When the incumbent's score is cached, only the child's draw is fresh, with variance $\bar{w}_V(c)/(NR)$, and the incumbent's frozen error enters the comparison as a bias; that is the case treated in Proposition 5.

K.4 Choosing among T candidates

For a fixed set of candidates evaluated with independent execution draws, the returned candidate is $\hat{c} = \arg\max_{k \le T} \hat{J}_V(c_k)$. Under this assumption, the errors $\varepsilon_k = \hat{J}_V(c_k) - J_V(c_k)$ are independent, with variances $v_k = \bar{w}_V(c_k)/(NR)$. Selection turns these zero-mean errors into two systematic costs: *optimism*, where the winner's reported score overstates its true score, and *regret*, where the winner is not the truly best candidate.

Proposition 4 (Selection optimism)**.** *For T candidates with near-equal true scores and a common noise level $v = \bar{w}/(NR)$,*

$$\mathbb{E}\big[\hat{J}_V(\hat{c}) - J_V(\hat{c})\big] \;\approx\; m_T \sqrt{\frac{\bar{w}}{NR}}, \qquad m_T = \mathbb{E}\big[\max_{k \le T} Z_k\big],\; Z_k \overset{iid}{\sim} \mathcal{N}(0,1), \tag{17}$$

where $m_5 \approx 1.16$, $m_{20} \approx 1.87$, $m_{100} \approx 2.51$, and $m_T/\sqrt{2\ln T} \to 1$ as $T \to \infty$ (David & Nagaraja, 2003). Keeping the optimism below a tolerance ϵ requires $R \ge m_T^2\, \bar{w}/(N\epsilon^2)$.

This is the optimizer's curse (Smith & Winkler, 2006). It grows with the number of candidates compared and shrinks only as $1/\sqrt{R}$. Near-ties are a useful reference case when late candidates differ by little. This fixed-candidate approximation does not account for adaptive proposal dependence.

Corollary 1 (Picking the best)**.** *If the best candidate leads every other by at least δ on V, then by a union bound $\Pr[\hat{c}$ is best$] \ge 1 - (T-1)\,\Phi\big(-\delta\sqrt{NR/\bar{w}_\Delta}\big)$. For this to be at least $1-\alpha$ it suffices that*

$$R \;\ge\; \frac{z^2_{\alpha/(T-1)}\, \bar{w}_\Delta}{N\, \delta^2}, \tag{18}$$

which is Proposition 3 with a Bonferroni-corrected critical value (Dunn, 1961) that grows with T.

Table 46: Selection noise on the full validation set, HotpotQA, Qwen3-8B. Left: runs per item $R^\star$ for a 95%-reliable pairwise decision (Eq. 16, $\bar{w}_\Delta = 0.100$), by validation-set size N and true gap Δ_V in points of accuracy. Right: expected optimism of the returned candidate among T near-tied candidates at $R = 1$ (Eq. 17, $N = 300$), and the runs per item that bring it below one point.

$N \setminus \Delta_V$	0.25	0.5	1	2	4
100	433	109	28	7	2
300	145	37	10	3	1
1000	44	11	3	1	1

T	optimism at R=1	R for $\leq$1 pt
5	1.5	3
20	2.4	6
100	3.3	11

The gap δ is the indifference zone of classical ranking and selection (Bechhofer, 1954). Our model keeps the optimizer's own protocol of uniform reruns on every candidate. Adaptive schemes that spend reruns only where candidates are close, such as racing (Maron & Moore, 1993) and best-arm identification (Audibert & Bubeck, 2010), applied to prompt selection by Shi et al. (2024), need fewer runs for the same guarantee.

K.5 How many reruns a decision needs

Table 46 instantiates Propositions 3 and 4 with constants measured on HotpotQA (Yang et al., 2018) with Qwen3-8B (Qwen Team, 2025), served with vLLM (Kwon et al., 2023) and optimized by GEPA's reference implementation in DSPy: $\bar{w}_\Delta \approx 0.100$ (median over 24 candidate pairs), $\bar{w} = 0.051$, and $\alpha = 0.05$. One reflective step from the seed program changes remeasured accuracy by -1.5 to $+10.8$ points ($+3.2$ on average), and gaps of that size are resolved on the full 300-item set with one to five runs per item. Gaps of a point or less, typical of late-stage steps, need about 10 to 145. With $T = 20$ near-tied candidates and $R = 1$, the returned candidate's validation score overstates its true score by about $1.87 \times 1.3 \approx 2.4$ points, and bringing that below one point takes about 6 reruns. In the testbed's three GEPA runs, which return the best of about 20 pool candidates scored once each, the returned program's recorded score minus its three-rerun estimate was $+2.0$, -1.3 and $+3.4$ points, and in two of the three runs a candidate scoring higher on remeasurement by 2.1 and 2.7 points remained unreturned in the pool.

K.6 Headroom and progress

How much selection noise matters depends on the gains the optimizer must resolve. The following fix/break model describes one way their distribution can change with parent accuracy. Let $J_V^\star$ be the accuracy the metric can credit at best (for the exact-match metric of HotpotQA we use a proxy derived from the share of seed answers two LLM judges accept; this is a modeling assumption, not a certified upper bound on achievable accuracy), so that an incumbent at accuracy J has headroom $J_V^\star - J$. A proposal does two things to the program it rewrites: it *fixes* some of the items the parent fails, and it *breaks* some of the items the parent solves. If a proposal fixes a fraction A of the fixable failures and breaks a fraction B of the successes, its true gain is $\Delta = A\,(J_V^\star - J) - B\,J$. The fractions vary from proposal to proposal; writing a, b for their means and s_a, s_b for their standard deviations (their *spreads*), taking them independent, and taking the gain to be Normal given J (the evidence is given below, Figure 25),

$$\Delta \mid J \;\sim\; \mathcal{N}\big(\mu(J),\, \sigma(J)^2\big), \qquad \mu(J) = a\,(J_V^\star - J) - b\,J, \qquad \sigma(J)^2 = s_a^2\,(J_V^\star - J)^2 + s_b^2\,J^2. \tag{19}$$

With nonnegative mean fix and break rates, the mean gain declines with parent accuracy. The spread can shrink or grow depending on the two variance terms. The mean crosses zero at the *break-even accuracy*

$$J_{\mathrm{eq}} = \frac{a}{a+b}\,J_V^\star, \tag{20}$$

below the ceiling: beneath J_{eq} proposals help on average, above it they hurt. The upper tail of the gains shrinks with the headroom (the s_a term) while the lower tail grows with J (the s_b term): near the top, wins become rare and small, losses common and large. A proposal is truly worse than its parent with probability $\Phi(-\mu(J)/\sigma(J))$, which passes one half at J_{eq}. The rates themselves are bounded, $A, B \in [0, 1]$: a proposal cannot fix more than the failures that remain or break more than the successes it has. The simulations of Section K.7 therefore sample A and B directly, as

Normal rates truncated to $[0, 1]$, so that the gain's upside vanishes with the headroom while its downside grows with J; the Normal of Eq. 19 is the mid-run approximation of that distribution, and Proposition 5 and Corollary 2 hold for that Normal, not for the truncated sampler. Under the bounded rates the special case $b = s_b = 0$, a proposer that never breaks anything, recovers a fraction a of the remaining headroom in expectation and can only climb.

The optimizer scores the child on V, obtaining $J + \Delta + \varepsilon$ with $\varepsilon \sim \mathcal{N}(0, \tau^2)$, $\tau^2 = \bar{w}_V/(NR)$ (Proposition 1), and accepts it if that beats the incumbent's cached score $J + e$, where e is the noise the incumbent's own score carried when it was measured.

Proposition 5 (Expected true change per proposal)**.** *Conditional on the incumbent's cached noise e, with $s = \sqrt{\sigma^2 + \tau^2}$ and $m = (\mu - e)/s$,*

$$\begin{aligned} \Pr[\textit{accept}] &= \Phi(m), \\ \mathbb{E}[\Delta \mathbf{1}[\textit{accept}]] &= \mu\Phi(m) + \frac{\sigma^2}{s}\phi(m), \\ \mathbb{E}[\Delta \mid \textit{accept}] &= \mu + \frac{\sigma^2}{s}\frac{\phi(m)}{\Phi(m)}. \end{aligned} \tag{21}$$

The first term of the expected change is the mean gain times the acceptance probability; the second is a *selection bonus*: because the observed score is positively correlated with the true gain, accepted children are better than the average proposal, by an amount that grows with the spread σ and shrinks as the noise τ dilutes the correlation.

Corollary 2 (When noise can reverse progress)**.** *The expected true change per proposal is non-negative whenever $\mu(J) \geq 0$, at every noise level τ and every cached noise e. It is negative only if*

$$\mu(J) \;<\; -\frac{\sigma^2}{s}\,\frac{\phi(m)}{\Phi(m)}, \tag{22}$$

that is, only where proposals are worse than the incumbent in expectation by more than the selection bonus.

Under Eq. 19 the mean gain is nonnegative below J_{eq}, so up to the break-even accuracy noise cannot reverse the expected trajectory; what it does there is two other things. It *stalls* the run: the incumbent's cached score is a score that won a comparison, so its error e is biased upward; every later child must beat that cached score, and when true gains are small the winning margin is mostly noise, so e stays high, m falls, and the acceptance rate goes toward zero while true gains remain. And it *degrades single runs*: with probability $\Phi(-\mu/\sigma)$ a proposal is truly worse, and such a proposal is still accepted with probability $\Phi((\Delta - e)/\tau)$; late in a run, where μ is small, accepted losses remain possible. Above J_{eq} the mean gain is negative and Eq. 22 can hold: a noisy run then drifts down in expectation, whereas a noise-free run, which accepts only true improvements, still inches up on the rare good proposals. Where a real proposer's break-even accuracy lies, relative to the accuracy the optimizer reaches within its budget, is an empirical question (Section K.7).

The scale on which noise competes with progress follows from the mean. Write $\kappa = a + b$ and let $H = J_{\mathrm{eq}} - J_V(c_0)$ be the seed's distance to break-even. For a heuristic recursion that applies the unselected mean proposal at each step, after k steps the distance is $H(1 - \kappa)^k$ and the next mean gain is $\mu_k = \kappa H(1 - \kappa)^k$. Substituting μ_k for Δ_V in Proposition 3 gives

$$R_k^\star \;=\; \frac{z^2\,\bar{w}_\Delta}{N\,\kappa^2 H^2}\,(1 - \kappa)^{-2k}, \tag{23}$$

so this heuristic resolution requirement grows geometrically. Accepted edits have a selection-adjusted mean, so this recursion is not an exact trajectory of the optimizer. Define the noise-to-progress ratio $\eta_k = z\sqrt{\bar{w}_\Delta/(NR)}\,/\,\mu_k$ at the runs per item R actually used, so that $\eta_k \leq 1$ is $R \geq R_k^\star$. The unselected mean step meets this resolution criterion while $\eta_k \leq 1$, which holds up to the *noise horizon*

$$k^\star \;=\; \frac{\ln\!\big(\kappa H\sqrt{NR}\,/\,(z\sqrt{\bar{w}_\Delta})\big)}{-\ln(1 - \kappa)}. \tag{24}$$

Beyond this heuristic $k^\star$, the unselected mean step is smaller than the chosen resolution threshold. Useful tail proposals can still occur; the expression is not a stopping-time bound. Reruns extend the

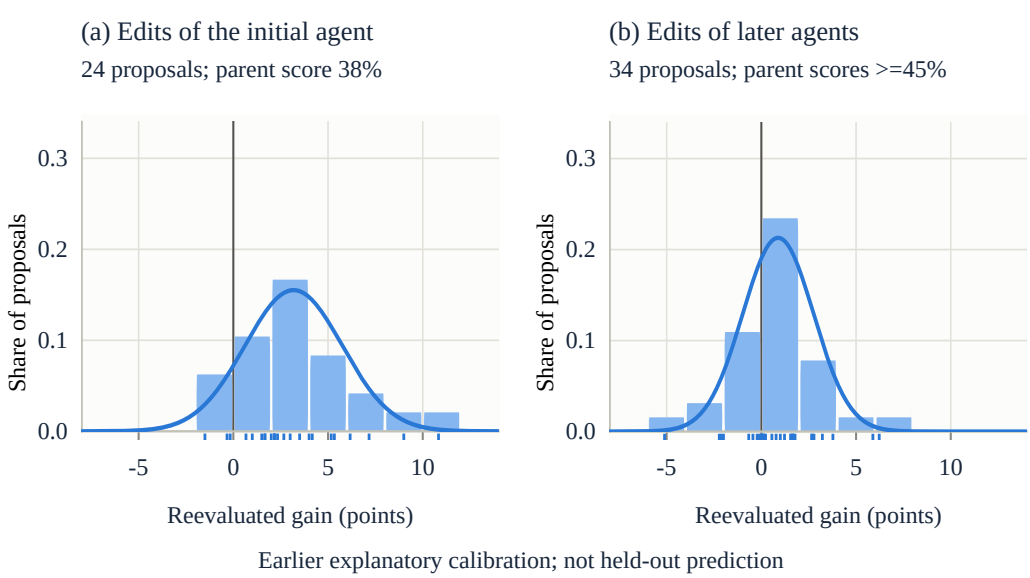

Figure 25: **Reevaluated proposal gains in the earlier Qwen3-8B/HotpotQA testbed.** (a) The 24 edits of the initial program, proposed using random three-item training minibatches and each evaluated on 300 held-out items with two repetitions and paired against that program. (b) Proposals retained in GEPA pools whose parents score at least 45%, paired against reevaluated parents with three repetitions; two gains beyond ± 15 points are omitted here but shown in Figure 26. Bars are shares per two-point bin; small ticks mark proposals. Curves retain the original Normal summaries with noise-corrected spreads. Gains remain estimates, not noise-free observations. This explanatory calibration is separate from held-out prediction.

horizon only logarithmically: each fourfold increase in R adds $\ln 2/(-\ln(1-\kappa))$ steps. The same trade-off between re-evaluating candidates and exploring new ones is familiar from evolutionary optimization under noisy fitness (Jin & Branke, 2005). The model separates the knobs:

- the base model sets the headroom and the rollout noise $\bar{w}_\Delta$;
- the proposer sets a, b, s_a and s_b, hence J_{eq} and the pace κ;
- the benchmark sets the validation size N;
- the practitioner sets the reruns R.

A base model that starts closer to its ceiling has smaller mean gains and, unless its runs are also more deterministic, reaches its noise horizon sooner.

K.7 Fitting the gain model to GEPA's proposals

Eq. 19 has four parameters and a shape; we measured the gain at two stages of a run to fix both (Figure 25). At the seed, 24 children of the seed program have estimated gains with mean $+3.2$ points and spread 2.6 after removing measurement error; 12% are below the parent, none significantly. A Normal is not rejected (Shapiro–Wilk $p = 0.11$), and the visible right skew is a mixture over which module was rewritten (answer-module rewrites $+5.5 \pm 4.4$, query-module rewrites $+1.9 \pm 0.7$). Along the runs, the 60 children GEPA itself added to its pool in three runs (parents from 35 to 61%) look different: the mean gain is $+1.4$ points for parents below 45%, $+1.6$ for parents at 45–54% and 0.0 above 54%, where the spread collapses to 0.4 points; overall 37% are worse than their parent, 5% significantly so, and 23% significantly better. Three measured effects are extreme, $+17$, $+16$ and -20, and are excluded from this exploratory calibration. That exclusion changes the estimated tails; a non-rejected Normality test does not validate the Gaussian model.

Fitting the bounded rates to the three measured levels, $+3.2 \pm 2.6$ at 38% (the seed's accuracy on the held-out items; it is 39.9 on V), $+1.6 \pm 2.4$ at 50% and 0.0 ± 0.4 at 58%, gives a fix rate A

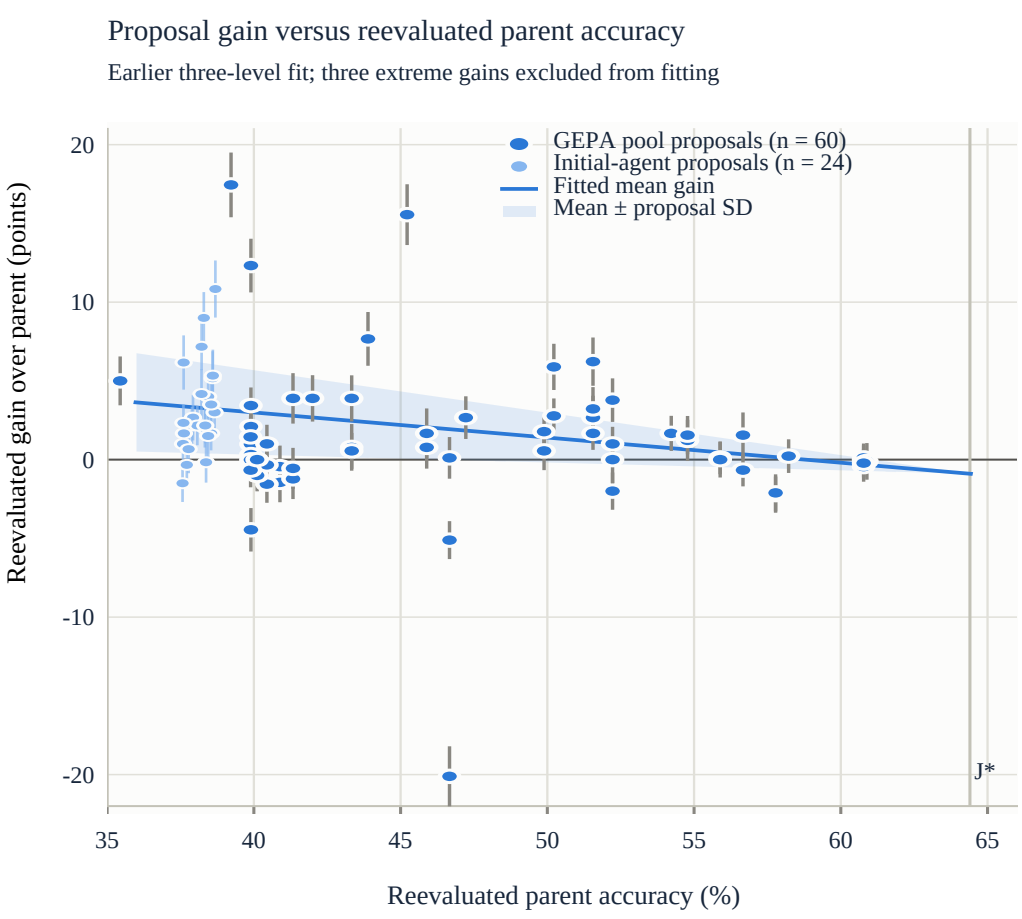

Figure 26: **Proposal gains decrease as parent accuracy rises in the earlier testbed fit.** All measured proposals are plotted against their parents' reevaluated accuracy, with ± 1 standard-error bars; proposals from the initial program are jittered horizontally. The line and band are the fitted mean and one proposal standard deviation from the earlier three-level calibration, not uncertainty in the fitted mean. Three gains beyond ± 15 points remain visible but are excluded from fitting. The vertical reference is the model's J^* parameter. These measurements do not validate the main paper's two-parent forecast.

with mean 0.146 and spread 0.110 (a half-Normal: most rewrites fix little, a few fix much) and a break rate B of 0.014 with no spread, which reproduce the three levels as $+3.3 \pm 2.9$, $+1.4 \pm 1.7$ and $+0.1 \pm 0.6$. Under them (Eq. 19 with these moments) the mean gain is $+3.0$, $+1.4$ and -0.2 points and the spread 2.7, 1.7 and 0.5 points at 40, 50 and 60%; the share of harmful proposals rises from 13% to 65%, and the break-even accuracy is $J_{\text{eq}} = 59\%$, five points below the ceiling. Both moments fall with the headroom, the spread faster than the mean: along a run the proposer turns into one that makes small, consistent proposals, the regime in which the accept test is hardest, and only near the ceiling do its proposals hurt on average.

Relation to the current prediction model. The bounded fix/break calibration above is an explanatory testbed model. It uses proposal measurements from the same three runs and is not held-out forecasting evidence. The main paper instead fits a linear conditional mean and shared spread from two parents in a reserved run, then evaluates separate runs (Appendix J). Its completed Qwen check is reported in Appendix J.6. Illustrative simulations of the current model are in Appendix J.4. These analyses use different calibrations and should not be pooled into a single accuracy or coverage claim.

L Supplementary Trace Summaries

Identity-count summaries (life table, kinds, and operation distributions) retain the first 137-run annotation batch because the two labellers use different granularity. Item-specific shares are shown separately by batch in Figure 28; code-level and edit-level analyses elsewhere use the expanded cohort. They provide context for the operator-specific analyses; retention and lineage do not by themselves establish performance effects.

L.1 The Life Table

Table 47 summarizes the number of interventions proposed, retained at any point, and present in the returned agent. Runs explore more interventions than their final agents contain. Retention is an annotation of candidate content, not evidence of an intervention's individual benefit. These descriptive summaries do not control for run length, task mix, or edit scope.

Table 47: **The life of an intervention in the first batch**, by method. Medians over runs unless a row is a share.

	GEPA	Meta-Harness	AdaEvolve	CRO
edits per run	101	28	219	16
interventions per run	94	29	102	13
interventions per edit	5	4	2	2
attempts per intervention (mean)	8.0	4.2	4.4	1.7
attempted once, never again	43%	36%	36%	63%
ever in a kept candidate	69%	47%	16%	19%
in the agent the run returns	10%	23%	6%	11%
interventions in the returned agent	11	5	5	2
item-specific	24%	3%	1%	0%

L.2 Kinds of Intervention

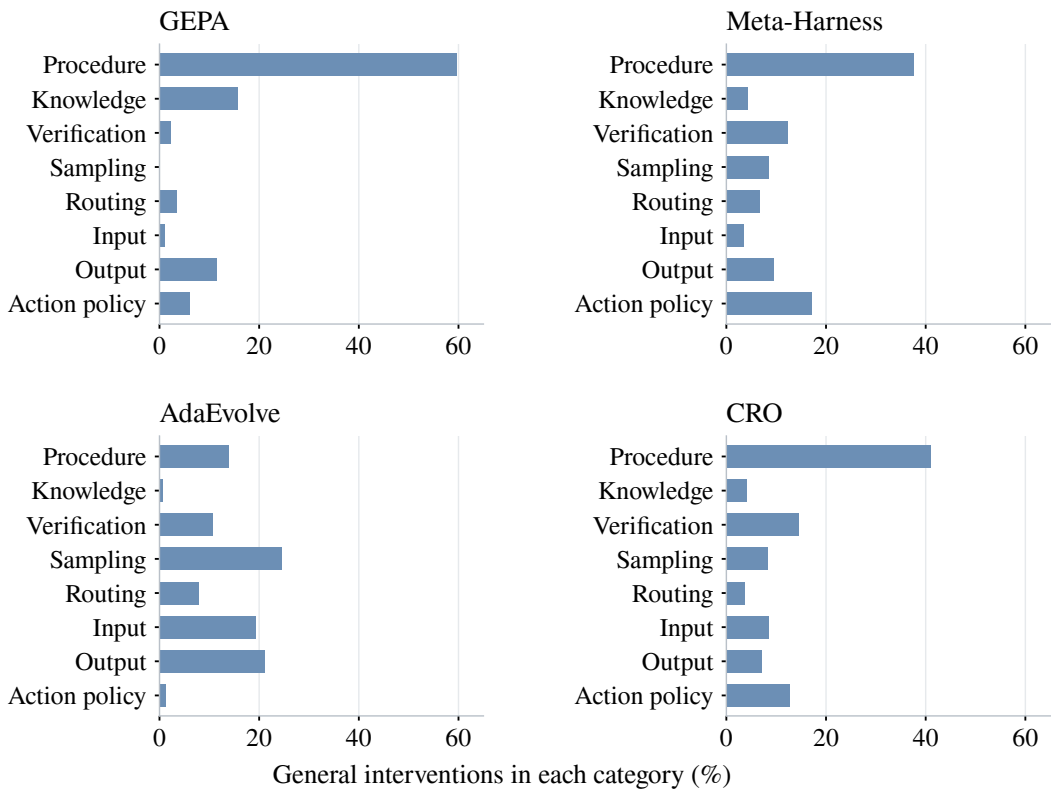

Figure 27: **The kinds of interventions each method proposes in the first batch.** Fractions of non-item-specific interventions assigned to each broad category, with optimizer models pooled. Categories group the task-level annotations (Appendix D). Runs require at least nine annotated edits, unlike Figure 29, which includes shorter runs. This yields 1,197 versus 1,203 Meta-Harness edits and 675 versus 739 CRO edits.

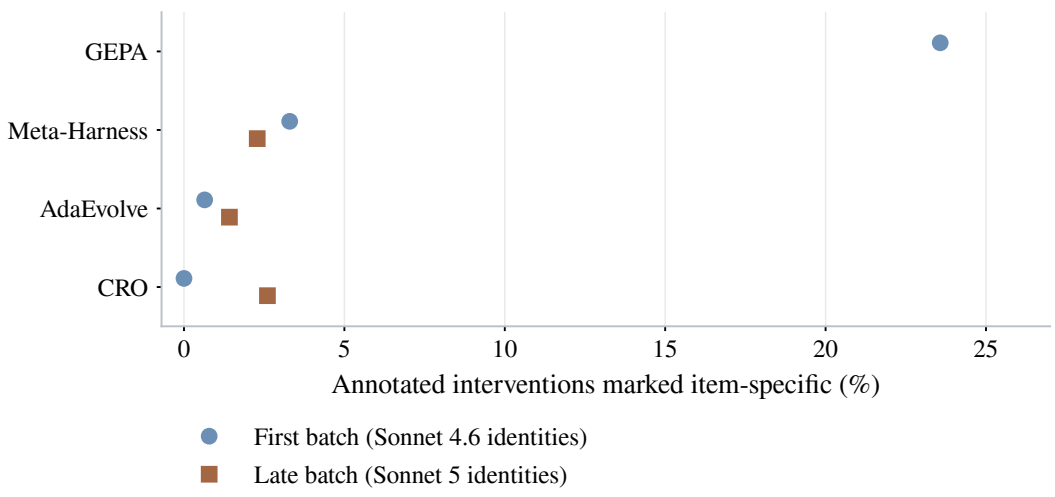

Figure 28: **Interventions specific to individual items.** Percentage of annotated interventions marked item-specific, by method and annotation batch. All Luna runs are included, shown separately for the first and late identity labellers. The same Sonnet 4.6 item-specificity rubric is used in both, but the intervention-weighted denominators remain sensitive to identity granularity. No late GEPA runs are available.

L.3 Runs as Sequences of Edits

Figure 29 separates introducing, developing, reintroducing, and removing interventions. The profiles differ across methods, but instructions, scope, tasks, and run lengths also differ. A reintroduction can arise when a method edits an earlier parent that lacks an intervention already tried elsewhere in the run; it need not repeat a failed implementation.

Figure 30 gives seven illustrative sequences. Acceptance, promotion, and measured improvement are separate events. These selected examples make that distinction visible but do not estimate method-level frequencies; the pooled comparisons are in Appendix G.

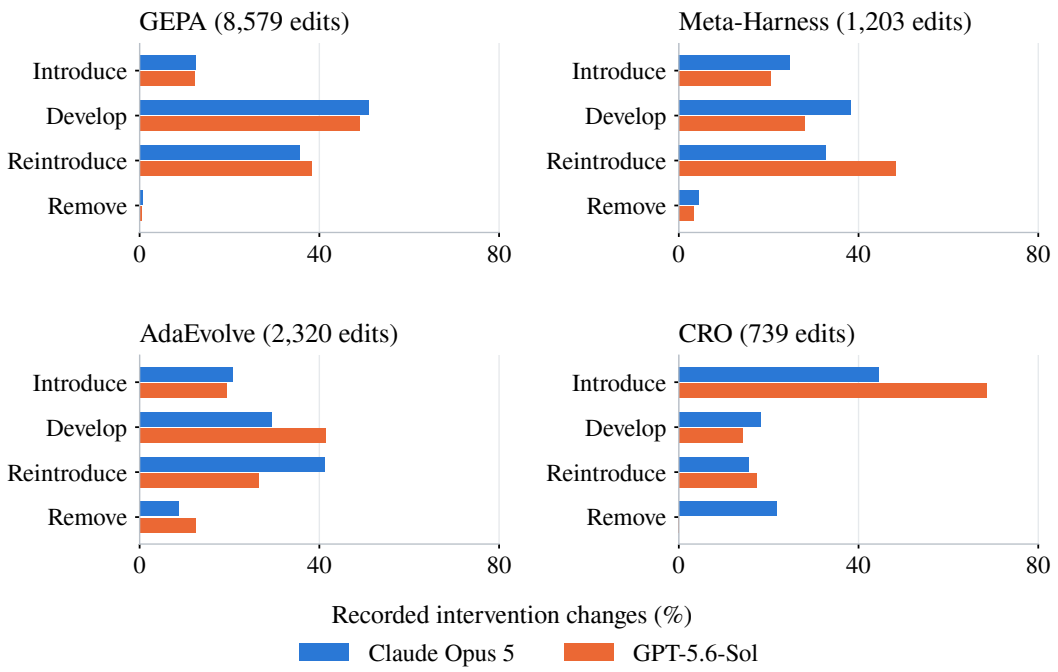

Figure 29: **How edits change interventions.** Share of recorded intervention changes that introduce, develop, reintroduce, or remove an intervention, by method and optimizer. One edit can change several interventions. Counts in titles are edits in the full first-batch cohort, including shorter runs omitted from Figure 27. Reintroduction may reuse an intervention absent from an earlier parent; it need not repeat a failed implementation.

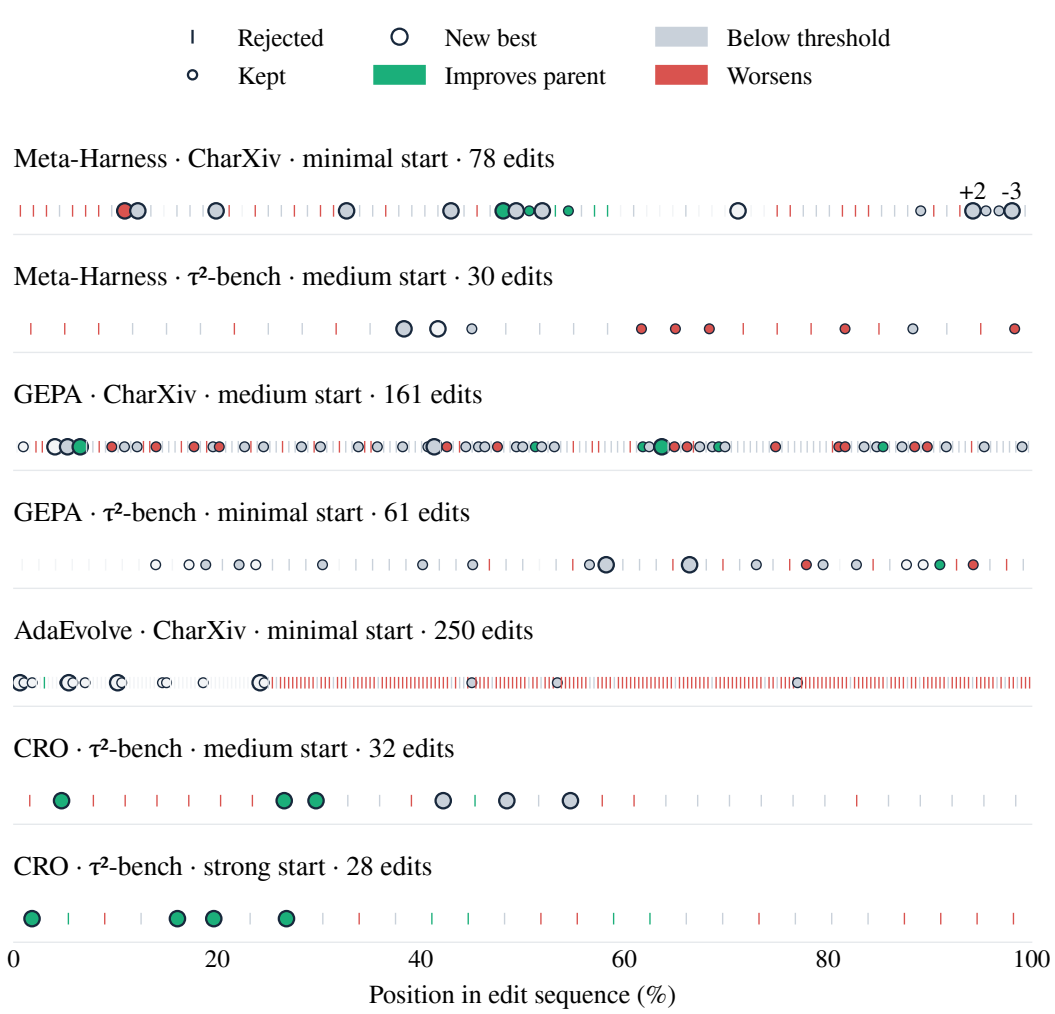

Figure 30: **Seven runs as sequences of edits.** A short vertical mark is a rejected edit; a small circle is kept; a large circle becomes the run's best. Color records measured change against the edited parent: improvement, below the one-standard-error threshold, or worsening. Numbers above new-best agents give their test change when a clean paired measurement exists. Horizontal position is normalized edit order, not elapsed time or cost. Light unclassified marks indicate unavailable effect measurements.

Figure 31 shows the operation mix over completed runs. Relative position aligns runs of different lengths and costs, so these curves describe the observed trajectories rather than a common amount of optimization effort. Meta-Harness shifts from introducing interventions to developing them within the first quarter of a run, and GEPA develops or reintroduces in most edits after the first tenth (reintroductions can reflect parents drawn from the Pareto front that lack an intervention the run has already tried). AdaEvolve's mix barely changes, and CRO keeps introducing to the end while removing more late in the run.

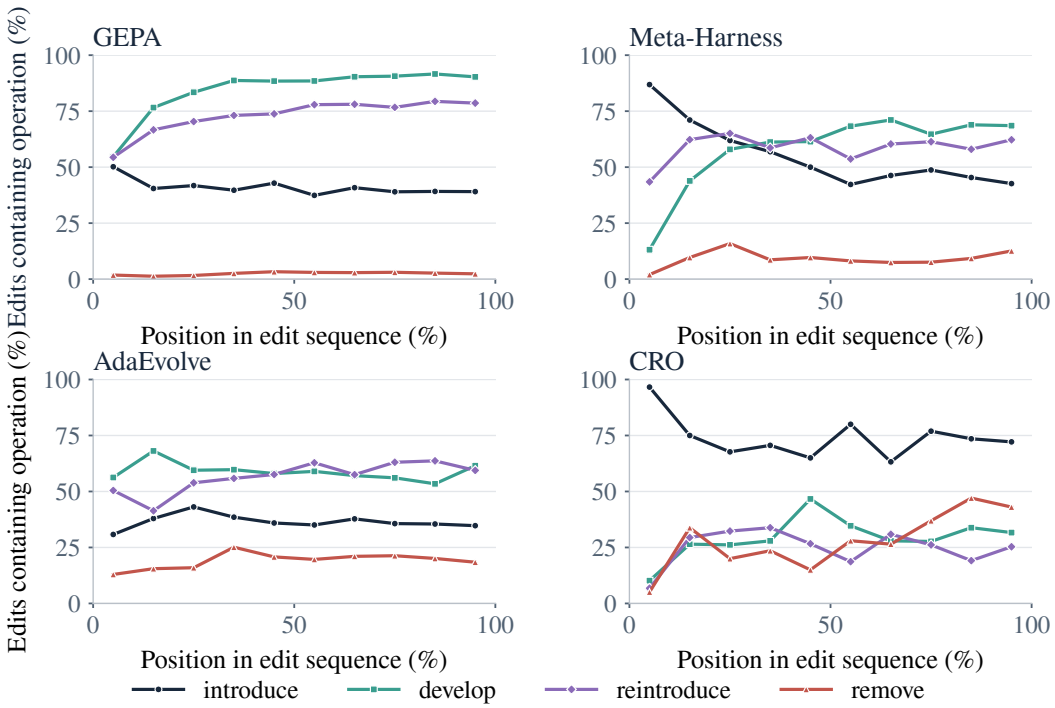

Figure 31: **Operations over the run.** Share of edits that contain each operation, by position in the run (ten equal bins), per method; an edit can contain several. First-batch runs on the frontier base model.