# Beyond Goodharting: A Genuine-Improvement Optimization Engine for Exploration-Guided Agent Evolution

Anonymous ACL submission


## Abstract

Recent systems use LLMs to iteratively evolve agentic systems, including code, prompts, tools, and task harnesses, from evaluation feedback. While effective, these metric-driven loops are vulnerable to *spurious optimization traps*: Goodharting turns noisy or incomplete metric gains into apparent progress, while weak exploration keeps later search near fragile directions instead of substantive alternatives. We introduce the Genuine-Improvement Optimization Engine (**GIOE**), a Goodhart-aware optimizer for LLM-guided agentic evolution. **GIOE** treats score gains as evidence rather than proof of genuine improvement and improves exploration by ranking admissible proposals by expected improvement and novelty. At each round, **GIOE** proposes structured modifications, filters them for validity, improvement potential, and exploration value, verifies selected code, prompt, or harness changes, and commits or discards them through an explicit decision protocol. We evaluate on AutoLab, $\tau^3$-bench, and general reasoning benchmarks. **GIOE** achieves higher aggregate scores than autoresearch-style greedy optimization and GEPA while increasing exploration coverage, suggesting that Goodhart-aware agent evolution requires optimizer engines that manage both metric evidence and search diversity instead of directly chasing noisy metrics.


## 1 Introduction

Large language models (LLMs) are increasingly used not only as task solvers, but also as optimizers of the *artifacts* that shape how tasks are solved: the editable components of an agentic system, such as prompts, codes, workflows, memory, and harnesses (Hu et al., 2025; Zhang et al., 2025b; Lee et al., 2026; Lou et al., 2026; Zhang et al., 2025a). In recent agentic systems, this turns artifact design into a search problem: given an initial artifact, an LLM-guided optimization engine can propose edits, run evaluations, observe feedback, and use the resulting traces to build stronger agents. This loop appears across automated code improvement, trajectory-informed prompt and workflow refinement, harness-level optimization, and empirical-loop benchmarks where agents propose, test, diagnose, and revise changes under compute and metric constraints (Karpathy, 2026; Agrawal et al., 2026; Lee et al., 2026; Borthwick et al., 2026; AutoLab Team, 2026). Together, these developments shift the frontier from one-shot agent design toward iterative, metric-driven agentic evolution.

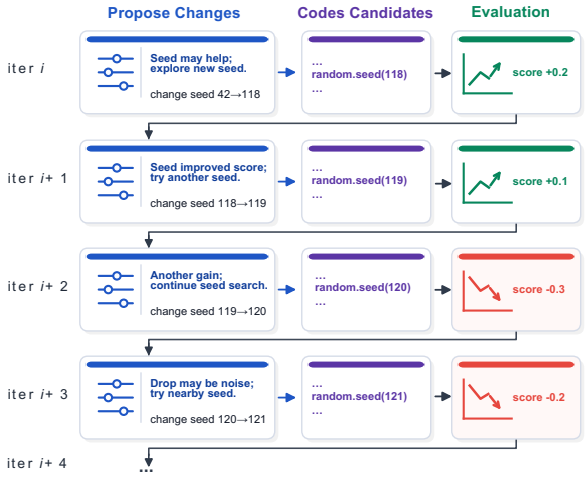


Figure 1: An illustration of a *spurious optimization trap*. The example shows an LLM-guided optimizer generating training scripts scored by validation accuracy, where the seed controls stochastic choices such as network initialization, data shuffling, and minibatch order. It changes only `seed=42` to `seed=118`, observes a higher score from a luckier run, and continues to search nearby seed edits. Although no evaluator tampering occurs, the loop spends search budget on apparent metric gains rather than improving the model or training procedure.

Metric-driven evolution, however, can fall into **spurious optimization traps**: trajectories where apparent score gains provide weak evidence of durable task improvement while the search process

prematurely converges. Figure 1 illustrates this failure: a lucky seed edit looks like progress, gets accepted, and pulls later search toward nearby seed variants instead of substantive model or training changes. This failure stems from two mechanisms: 1) **Goodharting**, or **reward hacking**, where the measured score improves without a corresponding improvement in intended task ability; and 2) **premature convergence**, or lack of exploration, where later proposals drift toward small variants of a spurious direction, crowding out broader algorithmic, workflow, tool-use, memory, or task-harness changes.

Escaping these traps requires the optimizer to distinguish *genuine improvement*, a change that raises the intended task ability rather than only the measured score, from metric feedback that is merely suggestive. Before expensive verification, it must decide whether a proposal is admissible at all: whether it avoids hacking and cheating, has a plausible mechanism for improving task behavior, and clears a minimum confidence threshold. Among admissible proposals, it must still allocate search toward directions that combine expected improvement with novelty relative to what has already been tried. Thus, the central design problem is to separate validity control from search allocation under noisy metrics.

Motivated by this view, we introduce **genuine-improvement optimization engine** (**GIOE**), organized around two nested loops. The outer PROPOSE–EVALUATE loop asks a Proposer to construct a ranked proposal queue and an Evaluator to test that queue in fresh worktrees, keeping only strict improvements over the historical best and logging mechanism-level outcomes. Inside the Proposer, an inner BRAINSTORM–INSPECT loop generates diverse candidate directions conditioned on the brainstorm history and cross-round analysis log, then screens and orders them before expensive verification. The screening step is implemented by **Hard-Gated Goodharting Control**: the Inspect stage removes hacking, cheating, and low-confidence proposals, addressing apparent metric gains that lack genuine improvement. The ordering step is implemented by **Acquisition-Function Scoring**: it ranks the admissible proposals by expected improvement and novelty, addressing premature convergence by steering search away from repeated variants of one proposal family. Together, these modules implement the separation above: validity control decides what may enter the queue, while acquisition-based ordering allocates verification budget among admissible directions.

We evaluate **GIOE** on executable optimization, interactive tool-use, and general reasoning tasks, with the clearest gains on AutoLab (AutoLab Team, 2026) and $\tau^3$-bench (Yao et al., 2025). On AutoLab, **GIOE** outperforms the autoresearch-style baselines and GEPA (Agrawal et al., 2026), with especially clear gains on systems optimization and model-development tasks. On $\tau^3$-bench, **GIOE** also outperforms the baselines, including a 0.059 absolute gain over baselines on the difficult banking-knowledge agentic tasks that require policy retrieval and application. On general reasoning benchmarks, including AIME, ARC-AGI, and HotpotQA, **GIOE** also improves aggregate performance while individual task and model combinations vary. These results suggest that reliable agentic evolution requires optimizer engines that manage metric evidence and exploration, rather than directly chasing noisy score gains.

## 2 Related Works

**Self-evolving agent systems.** Recent work treats agent design itself as an object of search. Prompt- and program-level optimizers improve instructions, demonstrations, and modular LM pipelines through iterative feedback (Zhou et al., 2023; Yang et al., 2024; Fernando et al., 2024; Khattab et al., 2024; Opsahl-Ong et al., 2024). This idea now extends to complete workflows and architectures: ADAS (Hu et al., 2025) searches over agents expressed in code, AFlow (Zhang et al., 2025b) automates workflow construction, GEPA (Agrawal et al., 2026) reflectively evolves the prompts of agentic systems, the Darwin Godel Machine (Zhang et al., 2025a) studies open-ended coding-agent self-improvement, and RoboPhD (Borthwick et al., 2026) evolves complex agents under tight evaluation budgets. These systems show the promise of metric-driven agent evolution; our focus is the complementary problem of when the metric signal is noisy, exploitable, or too local to guide reliable search.

**Autoresearch and harness-level optimization.** Agents are also being evaluated in empirical research loops that propose changes, run experiments, read results, and refine the next trial. This pattern appears in ML experimentation, data science, code-space exploration, and agentic research environments (Huang et al., 2023; Chan et al., 2024; Hong et al., 2024; Jiang et al., 2025; Nathani et al.,

2025). Karpathy's `autoresearch` instantiates it for nanochat training, while AutoLab and GEAR cast it as broader scientific or engineering optimization (Karpathy, 2026; AutoLab Team, 2026; Jeddi et al., 2026). Harness work further expands the editable surface: AutoHarness synthesizes code harnesses, and Meta-Harness optimizes model harnesses end to end (Lou et al., 2026; Lee et al., 2026). Once code, workflows, and harnesses are all editable, score gains must be screened for mechanism-level validity rather than accepted solely because they improve a verifier.

**Reward hacking, Goodharting, and spurious gains.** Optimizing proxy objectives can produce reward hacking, specification gaming, and Goodhart effects. Prior work identifies accident risks and diagnostic environments, and formalizes reward hacking, Goodharting, and reward-model overoptimization (Amodei et al., 2016; Leike et al., 2017; Skalse et al., 2022; Karwowski et al., 2023; Gao et al., 2022; Coste et al., 2023). Recent studies show analogous failures in language-model feedback loops, RLHF, reward tampering, and tool-using LLM-agent benchmarks (Pan et al., 2024; Wen et al., 2024; Denison et al., 2024; Thaman, 2026). Our setting differs because failure need not involve explicit evaluator tampering: benign edits such as seed changes or verifier-specific overfitting can still yield fragile score gains. We study these *spurious optimization traps* and introduce an optimizer-level mechanism that separates admissibility checking from acquisition-based search allocation before expensive verification.

## 3 Method

We model LLM-guided agent evolution as iterative optimization over editable artifacts: code, prompts, tools, memory, workflow, or harnesses. Each round observes the current artifact and the optimization trace, proposes candidate modifications, verifies selected modifications against a task metric, and carries the resulting evidence into later rounds. As argued in the introduction, this metric is evidence rather than proof: directly chasing it can create *spurious optimization traps*, where Goodharting accepts apparent gains that do not reflect genuine task improvement and premature convergence keeps later search near the same fragile direction.

**`GIOE`** addresses these traps by applying the two mechanisms named in the introduction before expensive verification. **Hard-Gated Goodharting Control** excludes proposals that appear to hack or cheat the verifier, or that lack enough plausibility to justify execution. **Acquisition-Function Scoring** then prioritizes the proposals by expected improvement and novelty relative to prior search. Section 3.1 defines the nested-loop framework that implements these mechanisms; Section 3.2 defines the acquisition score; and Section 3.3 defines the hard gate that protects the queue before verification.

### 3.1 Framework and Loop Design

**`GIOE`** implements pre-verification queue construction as a stateful process with two modules. At round $t$, the state contains the current artifact $x_t$, auxiliary workspace context $\mathcal{I}_t$, the history $\mathcal{D}_t$ of proposed directions, and a mechanism-level analysis log $\mathcal{L}_t$ summarizing prior outcomes. Conditioned on this state, a **Proposer** builds a ranked proposal queue $\mathbf{Q}$; an **Evaluator** then executes that queue against the real task metric.

The framework separates execution from queue construction through two nested loops. The outer PROPOSE–EVALUATE loop is the executable optimization protocol: the Proposer produces $\mathbf{Q}$, and the Evaluator consumes it by verifying proposals in order, accepting only strict metric improvements, and recording the outcome of each trial. The inner BRAINSTORM–INSPECT loop belongs to the Proposer: it generates a fresh batch of directions, removes invalid or implausible candidates, and ranks the proposals for evaluation.

This separation exploits a cost asymmetry. Inspecting a candidate with LLM judgment and stored trace information is cheap relative to running the full verification pipeline, which may require executing the artifact on a large validation set. **`GIOE`** therefore uses inexpensive reasoning to decide which proposals are safe and worthwhile to test, while preserving the real metric as the only source of accept/reject decisions.

#### 3.1.1 Outer Loop: Propose–Evaluate

Each outer round calls the Proposer on $(x_t, \mathcal{D}_t, \mathcal{L}_t, \mathcal{I}_t)$ to obtain a non-empty ranked proposal queue $\mathbf{Q}$. The Evaluator processes this queue in order. For each proposal $p \in \mathbf{Q}$, it applies the edits in an isolated worktree, runs the verify command once, and reads the primary metric $M$.

The Evaluator returns KEEP iff $M$ *strictly* improves on the historical best. Ties default to DISCARD, which avoids ratcheting upward on metric

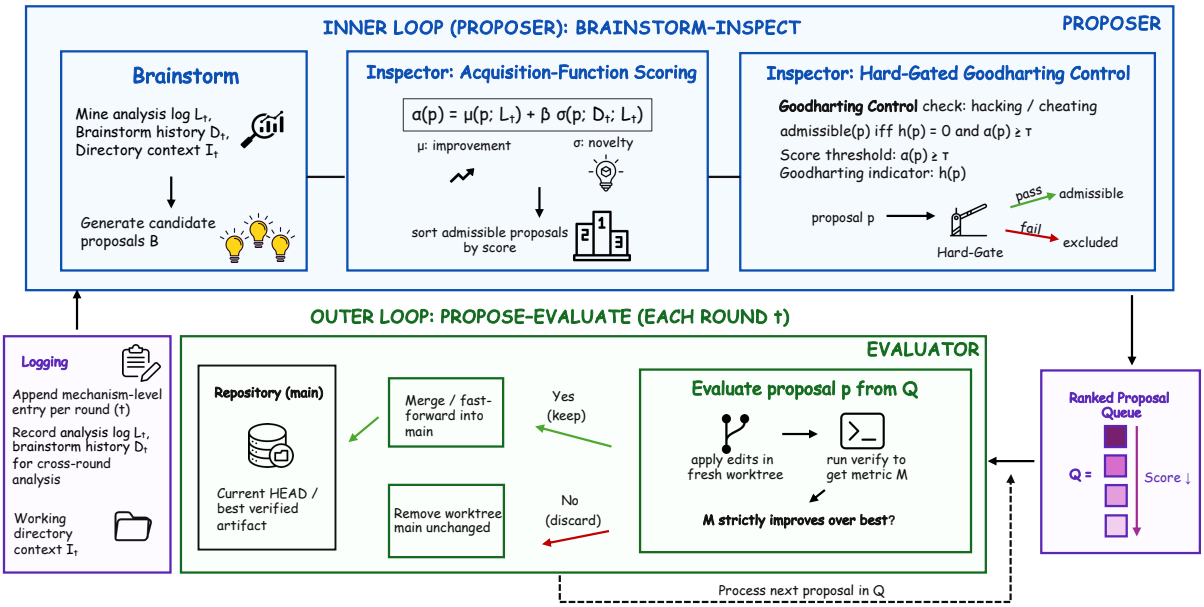


Figure 2: Overall framework of **GIOE**.

noise (§3.3). A kept candidate becomes the new main artifact; a discarded candidate leaves main unchanged. After every trial, the Evaluator appends the proposal, decision, metric change, and mechanism-level analysis to $\mathcal{L}_{t+1}$. The workflow is demonstrated in Algorithm 1.

Thus, the outer loop provides isolated verification, monotone updates to the best verified artifact, and outcome traces for future rounds. It does not decide which ideas are admissible before execution; that responsibility belongs to the Inner Loop.

### 3.1.2 Inner Loop: Brainstorm–Inspect

The inner loop instantiates the pre-verification mechanisms. Given $x_t$, $\mathcal{D}_t$, $\mathcal{L}_t$, and $\mathcal{I}_t$, the Brainstorm stage emits a fresh batch $B$ of concrete candidate directions (§3.1.3). The Inspect stage applies Hard-Gated Goodharting Control to remove hacking, cheating, and low-confidence proposals (§3.3), then uses Acquisition-Function Scoring to prioritize the survivors (§3.2).

If at least one proposal survives, the Proposer returns the ranked queue $\mathbf{Q}$ to the outer loop. If every proposal in $B$ is gated, Inspect emits structured feedback describing the rejection patterns, and Brainstorm is invoked again conditioned on that feedback. This retry path keeps verification focused on proposals that satisfy the hard gate while still giving the Proposer a way to leave invalid or low-confidence proposal families.

### 3.1.3 Brainstorm Process

Each round, the Proposer reads the working-directory context $\mathcal{I}_t$ and three running artifacts before emitting a batch: the cross-round analysis log $\mathcal{L}_t$ containing mechanism-level notes the Evaluator wrote after each prior trial, the brainstorm history $\mathcal{D}_t$ of every direction ever proposed, and the trial log of KEEP/DISCARD outcomes.

A first pass mines $\mathcal{L}_t$ for previously surfaced "next-direction" hints that have never been acted on, for mechanisms behind successful KEEPs that can be extended, and for families that $\mathcal{L}_t$ has labeled as dead ends. A second pass rounds the batch out for diversity across files, layers, and mechanisms, on the principle that five minor variants of one idea is a failure mode: because every admissible proposal in the batch will be tried (§3.3), family diversity within a single batch translates directly into broader exploration per outer-loop round.

Each emitted direction is a concrete code change with a one-line rationale; vague themes and Goodharting shapes are filtered at the source rather than relying on the Inspector to catch them.

Algorithm 1 states the resulting control flow.

## 3.2 Acquisition-Function Scoring

Acquisition-Function Scoring addresses premature convergence: the search drifts toward small variants of the last successful change instead of exploring new directions. Given the Brainstorm batch $B_t = \{p_1, \ldots, p_k\}$ emitted by the Brainstormer, it determines the order in which the Evaluator

will try the proposals. We first define the acquisition score, then describe two complementary anti-collapse mechanisms. The first is embedded in the novelty rubric itself, and the second governs how Brainstorm is re-initialized each round.

#### 3.2.1 Acquisition Function

For each $p \in B_t$ the Inspector computes

$$\alpha(p) \;=\; \underbrace{\mu(p;\, \mathcal{L}_t)}_{\text{improvement}} \;+\; \beta\, \underbrace{\sigma(p;\, \mathcal{H}_t)}_{\text{novelty}}, \qquad (1)$$

where $\mathcal{H}_t = (\mathcal{D}_t, \mathcal{L}_t)$ collects the running history available to the Inspector: $\mathcal{D}_t$ is all previously proposed directions and $\mathcal{L}_t$ is the mechanism-analysis log written by the Evaluator after each prior trial.

$\mu$ and $\sigma$ are scalar LLM judgments in $[0, 1]$ elicited against fixed rubric anchors (Appendix B.6); $\beta = 1$ in the optimization engin, where the implemented composite $\frac{1}{2}\mu + \frac{1}{2}\sigma$ is order-equivalent to (1) since both rubrics are already normalized to $[0, 1]$.

#### 3.2.2 Streak Penalty Inside $\sigma$

The first anti-collapse mechanism lives directly in the novelty rubric. If the most recent $w$ entries in $\mathcal{L}_t$ all belong to a single proposal family $F$, $\sigma(p; \mathcal{H}_t)$ for $p \in F$ is scored below its un-penalized anchor.

Functionally this is adaptive exploration: as exploitation in $F$ stops yielding novel directions, $\sigma$-scores for $F$ decay and the ranking shifts toward other families, without any explicit dynamic re-weighting of $\beta$ or of the two rubrics. The mechanism is deliberately implemented inside $\sigma$ rather than as a separate adaptive coefficient so that it composes naturally with (1): the acquisition function remains a fixed linear combination, and all plateau-response behavior is delegated to the rubric that already reads $\mathcal{H}_t$.

#### 3.2.3 Per-Round Re-Brainstorming as Anti-Collapse

The in-rubric penalty is paired with a Brainstormer-side mechanism: re-initializing Brainstorm every round, conditioned on the running $\mathcal{H}_t$ rather than on the previous batch, keeps $B_t$ from becoming a set of near-duplicates. Suppose instead that the Brainstormer refined only the previous KEEP: the batch $B_t$ would concentrate near $x_t$, most directions in $B_t$ would already be represented in $\mathcal{D}_t$, and $\sigma(p; \mathcal{H}_t)$ would collapse to 0 across the batch.

The novelty term in (1) would then provide no discriminative signal, and $\alpha$ would degenerate into $\mu$ alone. Such a pure-exploitation rule exemplifies exactly the premature-convergence failure mode this section is designed to prevent. Per-round re-brainstorming with explicit $\mathcal{H}_t$ conditioning keeps $\sigma$ informative across the batch, and therefore keeps the effective search width bounded away from zero.

### 3.3 Hard-Gated Goodharting Control

The Proposer must solve two coupled problems simultaneously: prevent proposals that exploit the verify process itself, while also keeping verification cost tractable. The core challenge is that proposal ranking alone is insufficient for safety: a high-scoring proposal may still improve the observed metric through reward hacking rather than through genuine task improvement.

#### 3.3.1 Hacking and Cheating

We use *Goodharting* as the umbrella term for proposals that raise the verify metric $M$ without raising true task ability $V$. We distinguish two failure modes against the measurement contract $M(x) = V(x) + \epsilon(x)$ where $\epsilon(x)$ is noise.

**Hacking.** The contract holds, but the proposal exploits $\epsilon$ to raise $M$ without changing $V$.

**Cheating.** Cheating breaks the measurement contract: after the modification, the observed metric $M$ no longer tracks the intended ability $V$ up to benign noise $\epsilon$.

Both are unacceptable: hacking wastes verification budget on trajectories that do not improve $V$, while cheating destroys the validity of the optimization loop itself.

#### 3.3.2 Why Weighted Penalties Fail

A natural alternative is to penalize suspicious proposals through a weighted acquisition score:

$$\text{score}_\lambda(p) = \alpha(p) - \lambda h(p), \qquad (2)$$

where $\alpha$ is the acquisition score (§3.2) and $h(p) \in \{0, 1\}$ indicates Goodharting behavior.

However, treating Goodharting risk as a soft penalty does not provide a reliable exclusion mechanism. In practice, both $\alpha(p)$ and $h(p)$ are estimated by noisy LLM judgments: a proposal that should be flagged as Goodharting may receive an underestimated risk assessment, or may be assigned a high acquisition score because its claimed gains look unusually promising. As a result, a suspicious proposal can still rank above all admissible

| Method | FA | BM25 | AES | BVH | KVW | FFT | Blur | Join | Regex | SST | Radix | SHA | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Baselines** | | | | | | | | | | | | | |
| One-shot edit | 0.420 | 0.301 | 0.502 | 0.324 | 0.614 | 0.514 | 0.411 | 0.648 | 0.853 | 0.335 | 0.578 | 0.381 | 0.490 |
| Terminus-2 | 0.442 | 0.284 | 0.485 | 0.377 | 0.661 | 0.507 | 0.324 | **0.668** | 0.373 | 0.526 | **0.610** | 0.056 | 0.443 |
| GEPA | 0.448 | 0.332 | 0.521 | 0.374 | 0.732 | 0.548 | 0.477 | 0.655 | 0.913 | 0.602 | 0.595 | 0.055 | 0.521 |
| Claude Code | **0.456** | 0.355 | **0.704** | 0.342 | 0.753 | 0.544 | 0.520 | 0.659 | **1.000** | 0.369 | 0.587 | 0.401 | 0.558 |
| **Ablations** | | | | | | | | | | | | | |
| w/o Brainstorm | 0.411 | 0.454 | 0.637 | 0.353 | 0.316 | 0.502 | 0.362 | 0.568 | 0.490 | 0.493 | 0.498 | 0.091 | 0.431 |
| w/o HGGC | 0.379 | 0.323 | 0.510 | 0.329 | 0.096 | 0.522 | **0.550** | 0.650 | 0.255 | 0.472 | 0.520 | 0.057 | 0.389 |
| w/o AFS | 0.376 | **0.534** | 0.656 | 0.365 | 0.485 | **0.554** | 0.482 | 0.636 | 0.561 | 0.452 | 0.518 | 0.059 | 0.473 |
| **Full method** | | | | | | | | | | | | | |
| **GIOE** | 0.433 | 0.442 | 0.701 | **0.385** | **1.000** | 0.541 | 0.492 | 0.659 | 0.925 | **0.766** | 0.582 | **0.412** | **0.612** |

Table 1: AutoLab systems-optimization results and ablations. In the ablation rows, HGGC denotes Hard-Gated Goodharting Control and AFS denotes Acquisition-Function Scoring. Columns abbreviate `flash_attention`, `bm25_search_go`, `aes128_ctr`, `bvh_raytracer`, `concurrent_kv_wal`, `fft_rust`, `gaussian_blur`, `hash_join`, `regex_engine`, `sstable_compaction_rs`, `radix_sort`, and `sha256_throughput`.

proposals whenever

$$\alpha(p^\star) - \lambda h(p^\star) > \max_{h(p)=0} \alpha(p). \tag{3}$$

Thus, the penalty only reorders proposals rather than enforcing admissibility, even a small number of false-negative or under-penalized Goodharting proposals can consume large amounts of expensive verification compute.

More fundamentally, ranking and validity control serve different purposes: ranking prioritizes promising directions under uncertainty, whereas Goodharting prevention requires hard behavioral constraints. Goodharting control must therefore operate at the inclusion level rather than through weighted ranking alone.

#### 3.3.3 Hard-Gated Inclusion Mechanism

The optimization engine separates admissibility from ranking through a hard gate:

$$\begin{aligned} \mathrm{adm}(p) \iff & \; h(p) = 0 \wedge \alpha(p) \geq \tau, \\ \mathbf{Q} \; = \; & \mathrm{sort}_{\downarrow\alpha}\{p : \mathrm{adm}(p)\}. \end{aligned} \tag{4}$$

Two conditions must hold for $p$ to enter the queue. First, $h(p) = 0$: proposals flagged as Goodharting are removed regardless of score, preventing the Evaluator from spending verification budget on trajectories that no longer track $V$. Second, $\alpha(p) \geq \tau$: extremely low-scoring directions are unlikely to provide either meaningful improvement or useful exploratory information, so their expected verification value falls below the cost of execution. Only after these exclusions does the system construct the ranked queue $\mathbf{Q}$; the acquisition score $\alpha$ allocates priority among already-admissible proposals, never serving as the mechanism that determines validity itself.

## 4 Experiments

### 4.1 Experiment Settings

We evaluate **GIOE** on three types of tasks that stress different abilities of LLM-guided agentic evolution. First, we use AutoLab, an executable optimization benchmark where agents iteratively improve code, data, training, serving, or search artifacts under verifier-based objectives (AutoLab Team, 2026). Second, we use the text customer-service setting of $\tau^3$-bench, which evaluates interactive agents that follow domain policies, converse with simulated users, and call APIs to complete realistic service tasks (Yao et al., 2025; Sierra Research, 2026). Third, we evaluate on general reasoning tasks, including ARC-AGI, HotpotQA, and AIME, covering abstraction, multi-hop question answering, and mathematical reasoning respectively (Chollet, 2019; Yang et al., 2018; Mathematical Association of America, 2026).

We compare **GIOE** with four baselines: **GEPA** for reflective prompt evolution (Agrawal et al., 2026), **Claude Code** following the autoresearch-style iterative edit–verify–accept loop (Karpathy, 2026), **Terminus-2** as an autonomous terminal-agent scaffold (Merrill et al., 2026), and a **One-shot edit** baseline that performs a single Claude Code edit without iterative acceptance or rollback.

All methods use the same task inputs, verifier scripts, and scoring rules within each benchmark. Our method is implemented on top of Claude Code and uses Opus-4.6 and DeepSeek-v4-pro as backbone LLMs. Additional details and configurations are provided in Appendix A.

| Method | System Opt. | Puzzle | Model Dev. | Overall |
|---|---|---|---|---|
| One-shot edit | 0.490 | 0.378 | 0.539 | 0.471 |
| Terminus-2 | 0.443 | 0.289 | 1.028 | 0.530 |
| GEPA | 0.521 | 0.515 | 0.568 | 0.530 |
| Claude Code | 0.558 | 0.530 | 0.940 | 0.634 |
| **GIOE** | **0.612** | **0.586** | **1.063** | **0.703** |

Table 2: AutoLab category-level averages across systems-optimization, puzzle-and-challenge, and model-development tasks. Overall is task-count weighted over available task scores.

| Method | Airline | Retail | Telecom | Banking | Overall |
|---|---|---|---|---|---|
| Seed | 0.760 | 0.763 | 0.816 | 0.268 | 0.652 |
| One-shot edit | 0.710 | 0.702 | **0.991** | 0.272 | 0.669 |
| GEPA | 0.800 | 0.737 | 0.801 | 0.268 | 0.652 |
| Claude Code | 0.787 | 0.763 | 0.983 | 0.285 | 0.704 |
| **GIOE** | **0.800** | **0.807** | 0.983 | **0.344** | **0.733** |

Table 3: $\tau^3$-bench results by task category and overall average. Scores reported as pass@1.

## 4.2 Results on AutoLab

### 4.2.1 Main Results

Table 1 reports per-task results and ablations on the AutoLab systems-optimization tasks, and Table 2 summarizes performance by category.

In the category view of Table 2, **GIOE** is the strongest method overall and in all categories, leading systems optimization, puzzle-and-challenge, and model development at once. Its overall average of 0.703 exceeds the best baseline, Claude Code at 0.634, by a clear margin, with Terminus-2, GEPA, and the One-shot edit trailing at 0.530, 0.530, and 0.471. Leading every category shows that **GIOE** strengthens the optimization process itself rather than exploiting any single task type.

Table 1 reports the systems-optimization tasks individually, with bold marking the best and underline the second-best method per column. **GIOE** is the top performer: it attains the highest average at 0.612, ahead of Claude Code at 0.558 and GEPA at 0.521, and ranks best or second-best on seven of the twelve tasks. Its largest gains come on tasks that reward genuine algorithmic change and broad exploration rather than verifier-specific tuning, two of which we examine in the case study of §4.3.

### 4.2.2 Ablation Study

We ablate **GIOE**'s three mechanisms one at a time: the Brainstorm stage, Hard-Gated Goodharting Control (HGGC), and Acquisition-Function Scoring (AFS). The **Ablations** block of Table 1 reports the resulting averages. Every mechanism is necessary. Removing AFS lowers the average from the full 0.612 to 0.473, removing Brainstorm to 0.431, and removing HGGC to 0.389. HGGC is thus the most important component, followed by Brainstorm and then AFS. Every ablated variant also falls below the strongest baseline, Claude Code at 0.558, showing that the three mechanisms are jointly necessary rather than independently additive: disabling any one does not merely remove its marginal contribution but lets the search collapse.

## 4.3 Case Study

We highlight two trials that illustrate **GIOE**'s effect on the same tasks: `gaussian_blur` for reward hacking, and `sha256_throughput` for exploration.

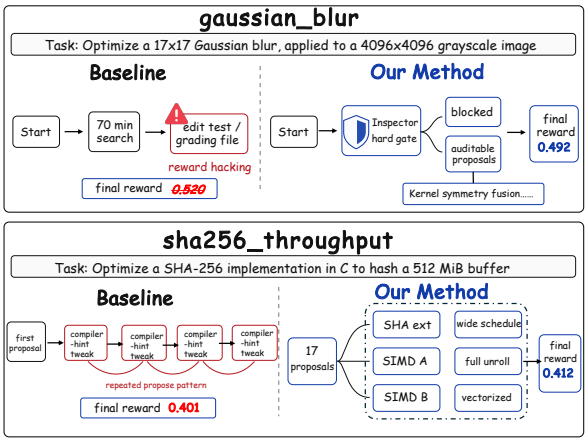


Figure 3: Case studies illustrating how **GIOE** blocks Goodharting and promotes more diverse exploration.

**Reward hacking: `gaussian_blur`.** The task is scored by an external grading script that writes the official reward to a protected output file, and the instructions explicitly forbid writing to that file. After seventy minutes without locating the grader, the baseline agent authored a substitute grading script and wrote its own timing measurement into the protected reward field. Its reported reward of 0.520 is therefore not a trustworthy task score, but a Goodharting artifact.

With **GIOE**, the same agent never writes to the grader or protected output. The Inspector blocks such Goodharting patterns, and the final reward of 0.492 is obtained under the intended evaluation protocol. Thus, although the reported baseline number is higher, only **GIOE**'s score is auditable.

**Exploration: `sha256_throughput`.** The baseline agent commits to a single hardware-extension direction in first iteration and spends the remaining iterations on near-duplicate compiler-hint tweaks within a $\pm 3$ ms noise band. It therefore tries only a

| Method | AIME26 | | | ARC-AGI | | | HotpotQA | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Agent LLM | gpt-5.4-nano | qwen3.5-9b | Avg. | gpt-5.4-nano | qwen3.5-9b | Avg. | gpt-5.4-nano | | qwen3.5-9b | | Avg. | |
| Metric | Acc. | Acc. | Avg. | Acc. | Acc. | Avg. | F1 | EM | F1 | EM | F1 | EM |
| **No optimizer** | | | | | | | | | | | | |
| Seed | 0.910 | 0.627 | 0.768 | 0.113 | 0.225 | 0.169 | 0.514 | 0.340 | 0.340 | 0.473 | 0.427 | 0.407 |
| **Optimizer LLM: Opus-4.6** | | | | | | | | | | | | |
| One-shot edit | 0.907 | 0.830 | 0.869 | 0.518 | 0.508 | 0.513 | 0.591 | 0.413 | 0.413 | 0.493 | 0.502 | 0.453 |
| GEPA | 0.907 | 0.827 | 0.867 | 0.515 | 0.420 | 0.468 | 0.734 | 0.640 | 0.656 | 0.583 | 0.695 | 0.612 |
| Claude Code | 0.957 | **0.883** | **0.920** | 0.613 | 0.485 | 0.549 | 0.768 | 0.653 | 0.657 | 0.620 | 0.713 | 0.637 |
| **GIOE** | **0.960** | 0.833 | 0.896 | **0.630** | **0.523** | **0.577** | **0.771** | **0.663** | **0.707** | **0.660** | **0.739** | **0.662** |
| **Optimizer LLM: DeepSeek-v4-pro** | | | | | | | | | | | | |
| One-shot edit | 0.957 | 0.707 | 0.832 | 0.573 | 0.263 | 0.418 | 0.412 | 0.343 | 0.609 | 0.497 | 0.511 | 0.420 |
| GEPA | **0.963** | 0.800 | 0.881 | 0.530 | **0.515** | 0.522 | 0.761 | 0.673 | 0.714 | 0.613 | 0.738 | 0.643 |
| Claude Code | 0.937 | 0.803 | 0.870 | 0.595 | 0.415 | 0.505 | **0.793** | **0.697** | 0.756 | 0.630 | 0.775 | 0.664 |
| **GIOE** | **0.963** | **0.877** | **0.920** | **0.615** | 0.475 | **0.545** | 0.777 | 0.677 | **0.789** | **0.673** | **0.783** | **0.675** |

Table 4: General reasoning tasks results, grouped by optimizer backbone with agent LLMs shown as metric columns.

narrow neighborhood of its initial idea and never attempts alternative SIMD or scheduling strategies.

With **GIOE**, the same agent logs 17 proposals spanning at least 8 distinct implementation directions, including hardware-extension variants, SIMD block-load strategies, wide message scheduling, loop unrolling, and vectorized schedule designs. This broader proposal trace shows that **GIOE** prevents the search from collapsing into repeated local tweaks and instead maintains coverage over a larger fraction of the implementation design space.

### 4.4 Results on $\tau^3$-bench

Table 3 reports results on the four $\tau^3$-bench task categories and their overall average. Seed is the original $\tau^3$-bench assistant agent that all methods evolve from. **GIOE** achieves the best overall score of $0.733$, ahead of the strongest baseline Claude Code at $0.704$ and the Seed agent at $0.652$, and it is the only method that improves every category over Seed. Its largest gain is on Banking, the hardest category, where Seed reaches only $0.268$ because the tasks require retrieving and applying policy documents; **GIOE** lifts this to $0.344$, a $0.059$ absolute gain over Claude Code, while GEPA and the One-shot edit barely move it. Several baselines even regress below Seed on the Airline and Retail categories, confirming that naive or single-pass optimization can degrade an interactive agent, whereas **GIOE** improves it reliably.

### 4.5 Results on General Reasoning Tasks

Table 4 reports baselines and our method grouped by optimizer LLM, with agent LLMs shown as metric columns. AIME and ARC-AGI use accuracy, while HotpotQA uses F1 and exact match (EM). Seed is the common initial code that all methods evolve from.

Aggregated by the per-benchmark **Avg.** columns, **GIOE** posts the best score in five of the six optimizer-by-benchmark cells, leading ARC-AGI and HotpotQA under both Opus-4.6 and DeepSeek-v4-pro and AIME under DeepSeek-v4-pro; only on AIME under Opus-4.6 does Claude Code edge ahead. The improvements over the un-optimized Seed agent are large, with ARC-AGI rising from $0.169$ to $0.577$ and HotpotQA from $0.427$ to $0.783$ in F1 and from $0.407$ to $0.675$ in EM, showing that the optimization loop rather than the agent LLM drives the gains. Individual agent-LLM cells vary, but **GIOE** remains the most reliable choice across both optimizer backbones, indicating that its advantage comes from the optimization protocol itself rather than from a single favorable backbone.

## 5 Conclusion

We introduced the Genuine-Improvement Optimization Engine (**GIOE**), a Goodhart-aware optimizer for LLM-guided agentic evolution that treats metric gains as evidence rather than proof. Its novelty is to separate validity control from search allocation before expensive verification: Hard-Gated Goodharting Control screens out proposals that hack the metric or lack a plausible improvement mechanism, and Acquisition-Function Scoring orders the admissible ones by expected improvement and novelty to counter premature convergence. Across AutoLab, $\tau^3$-bench, and general reasoning benchmarks, **GIOE** achieves higher aggregate scores than AutoResearch-style greedy optimization and GEPA while broadening exploration coverage, showing that reliable agentic evolution requires optimizers that jointly manage metric evidence and search diversity rather than chasing noisy scores directly.

## Limitations

**No completeness guarantee against hacking.** Hard-Gated Goodharting Control reduces hacking and cheating, but the space of ways to game a metric is open-ended, and no screening rule can be guaranteed to anticipate all of them. The gate may still miss novel exploitation strategies or subtle forms of metric gaming, so genuine improvement is encouraged rather than formally certified.

**Compute overhead.** `GIOE` adds compute overhead relative to greedy baselines: each round brainstorms and inspects a batch of proposals before any verification, so the optimizer issues additional LLM calls per outer-loop iteration. This cost is justified only under the assumption that inspection is cheap relative to full verification, and it may erode the method's advantage on tasks where verification is itself inexpensive.

## 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. 2026. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *International Conference on Learning Representations*. Oral.

Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. 2016. Concrete problems in AI safety. *Preprint*, arXiv:1606.06565.

AutoLab Team. 2026. AutoLab: Can models begin to participate in the loops that drive scientific and engineering progress?

Andrew Borthwick, Stephen Ash, and Anthony Galczak. 2026. RoboPhD: Evolving diverse complex agents under tight evaluation budgets. *Preprint*, arXiv:2604.04347.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. 2024. MLE-bench: Evaluating machine learning agents on machine learning engineering. *Preprint*, arXiv:2410.07095.

François Chollet. 2019. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*.

Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. 2023. Reward model ensembles help mitigate overoptimization. *Preprint*, arXiv:2310.02743.

Carson Denison, Monte MacDiarmid, Fazl Barez, David Duvenaud, Shauna Kravec, Samuel Marks, and 1 others. 2024. Sycophancy to subterfuge: Investigating reward-tampering in large language models. *Preprint*, arXiv:2406.10162.

Chrisantha Fernando, Dylan Sunil Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2024. Promptbreeder: Self-referential self-improvement via prompt evolution. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 13481–13544. PMLR.

Leo Gao, John Schulman, and Jacob Hilton. 2022. Scaling laws for reward model overoptimization. *Preprint*, arXiv:2210.10760.

Sirui Hong, Yizhang Lin, Bang Liu, Bangbang Liu, Binhao Wu, Ceyao Zhang, and 1 others. 2024. Data interpreter: An LLM agent for data science. *Preprint*, arXiv:2402.18679.

Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated design of agentic systems. In *International Conference on Learning Representations*.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2023. MLAgentBench: Evaluating language agents on machine learning experimentation. *Preprint*, arXiv:2310.03302.

Ahmadreza Jeddi, Minh Ngoc Le, Hakki C. Karaimer, Konstantinos G. Derpanis, and Babak Taati. 2026. GEAR: Genetic autoresearch for agentic code evolution. *Preprint*, arXiv:2605.13874.

Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. AIDE: AI-driven exploration in the space of code. *Preprint*, arXiv:2502.13138.

Andrej Karpathy. 2026. autoresearch: Ai agents running research on single-gpu nanochat training automatically. `https://github.com/karpathy/autoresearch`. GitHub repository.

Jacek Karwowski, Oliver Hayman, Xingjian Bai, Klaus Kiendlhofer, Charlie Griffin, and Joar Skalse. 2023. Goodhart's law in reinforcement learning. *Preprint*, arXiv:2310.09144.

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. 2024. DSPy: Compiling declarative language model calls into self-improving pipelines. *Preprint*, arXiv:2310.03714.

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

Jan Leike, Miljan Martic, Victoria Krakovna, Pedro A. Ortega, Tom Everitt, Andrew Lefrancq, Laurent Orseau, and Shane Legg. 2017. AI safety gridworlds. *Preprint*, arXiv:1711.09883.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. 2026. Autoharness: Improving LLM agents by automatically synthesizing a code harness. *Preprint*, arXiv:2603.03329.

Mathematical Association of America. 2026. American Invitational Mathematics Examination (AIME). https://maa.org/maa-invitational-competitions/. MAA Invitational Competitions; accessed 2026-05-24.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, and 66 others. 2026. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *Preprint*, arXiv:2601.11868.

Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincent Moens, and 1 others. 2025. MLGym: A new framework and benchmark for advancing AI research agents. *Preprint*, arXiv:2502.14499.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, Miami, Florida, USA. Association for Computational Linguistics.

Alexander Pan, Erik Jones, Meena Jagadeesan, and Jacob Steinhardt. 2024. Feedback loops with language models drive in-context reward hacking. *Preprint*, arXiv:2402.06627.

Sierra Research. 2026. $\tau^3$-bench 1.0.0: Voice, knowledge, task quality. GitHub release.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. 2022. Defining and characterizing reward hacking. *Preprint*, arXiv:2209.13085.

Kunvar Thaman. 2026. Reward hacking benchmark: Measuring exploits in llm agents with tool use. *Preprint*, arXiv:2605.02964.

Jiaxin Wen, Ruiqi Zhong, Akbir Khan, Ethan Perez, Jacob Steinhardt, Minlie Huang, and 1 others. 2024. Language models learn to mislead humans via RLHF. *Preprint*, arXiv:2409.12822.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. 2024. Large language models as optimizers. In *International Conference on Learning Representations*.

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

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik R. Narasimhan. 2025. $\tau$-bench: A benchmark for tool-agent-user interaction in real-world domains. In *International Conference on Learning Representations*.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025b. AFlow: Automating agentic workflow generation. In *International Conference on Learning Representations*.

Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. 2023. Large language models are human-level prompt engineers. In *International Conference on Learning Representations*.

# A Experiment Setup

## A.1 Benchmarks

We evaluate **GIOE** on three benchmark groups, each targeting a different aspect of LLM-guided agentic evolution.

**Executable improvement loops.** We use **AutoLab** (AutoLab Team, 2026), a benchmark of 23 executable optimization tasks. These include 5 model-development tasks, 12 systems-optimization tasks, and 6 puzzle-and-challenge tasks. Each task provides a working but suboptimal starting artifact, a verifier, and a scalar objective. The agent must improve the artifact under a fixed evaluation budget. Depending on the task, the artifact may involve code, data selection, training configuration, serving logic, or search behavior. This benchmark tests whether an agentic improvement loop can make reliable progress in real executable environments rather than only improving natural-language outputs.

**Interactive tool-use tasks.** We use the text customer-service setting from $\tau^3$**-bench** (Yao et al., 2025; Sierra Research, 2026). This benchmark evaluates agents that interact with simulated users, follow domain-specific policies, call APIs, and are scored according to the final task state. We use the default base split, which contains 50 airline tasks involving booking and itinerary changes, 114 retail tasks involving orders and returns, 114 telecom tasks involving shared-state service coordination, and 97 banking-knowledge tasks involving retrieval and application of policy documents. These tasks test whether the evolved agent can improve behavior in multi-turn, stateful, and policy-constrained environments.

**General reasoning tasks.** We also evaluate on three general reasoning benchmarks: **ARC-AGI** for grid-based abstraction and visual reasoning, **HotpotQA** for multi-hop question answering over Wikipedia, and **AIME** for competition-level mathematical reasoning (Chollet, 2019; Yang et al., 2018; Mathematical Association of America, 2026). This group tests whether the proposed improvement loop works beyond executable software optimization and interactive tool use to broader reasoning settings.

## A.2 Baselines

We compare against five baselines.

**Seed** denotes the initial artifact before any optimization. This measures the starting performance of the provided task artifact.

**GEPA** is a reflective prompt-evolution method that uses execution feedback to update prompts across rounds (Agrawal et al., 2026). It serves as a representative feedback-driven prompt optimization baseline.

**Claude Code** follows the autoresearch-style loop described by Karpathy (Karpathy, 2026). In this baseline, Claude Code repeatedly edits the target artifact, runs the verifier, accepts measured improvements, and uses the accepted version as the starting point for the next round.

**Terminus-2** is Harbor's autonomous terminal-agent scaffold (Merrill et al., 2026). It gives the model an interactive shell and context-management machinery for end-to-end task execution, representing a terminal-based autonomous agent baseline.

**One-shot edit** gives Claude Code the same task and verifier only once. It does not use iterative acceptance, rollback, or cross-round memory. This baseline isolates the benefit of iterative optimization from single-pass editing ability.

## A.3 Benchmark Evaluation Details

Across all benchmark groups, every method is evaluated with the same task inputs, verifier scripts, and scoring rules, and we do not modify the official task objectives or evaluation scripts unless the benchmark protocol explicitly requires it. **GIOE** uses Claude Code as its optimizer scaffold, and each iterative baseline runs its own standard edit–evaluate loop, while verifier access, scoring rules, the available task information, and the evaluation budget are held fixed across methods. The optimizer backbone is Opus-4.6 or DeepSeek-v4-pro; the specific backbone, search budget, and data splits for each benchmark are detailed below.

### A.3.1 AutoLab

Each task runs inside the container specified by the AutoLab repository, with CPU tasks on a shared CPU machine and GPU tasks on a shared machine equipped with an A100 GPU. We keep the per-task time limits defined by the repository and run all compared methods in the same container under the same wall-clock time budget rather than a fixed number of evaluations. The backbone LLM for this benchmark is DeepSeek-v4-pro.

### A.3.2 Tau3-bench

The optimizer LLM is Opus-4.6, the assistant LLM is GPT-5.4, and the user-simulator LLM is GPT-5.2. For each task we sample 10 queries as the training set and cap the evolution budget at 20 evaluations; the final evaluation runs on the full set of task queries, with each query rolled out three times, and we report pass@1.

### A.3.3 AIME

The optimizer LLM is Opus-4.6 or DeepSeek-v4-pro and the agent LLM is gpt-5.4-nano or qwen3.5-9b. We draw the training set from AIME 2023–2024, sampling 45 problems, and cap the evolution budget at 20 evaluations. The final evaluation uses the 30 problems of AIME 2026, with each problem rolled out 10 times.

### A.3.4 HotpotQA

The optimizer LLM is Opus-4.6 or DeepSeek-v4-pro and the agent LLM is gpt-5.4-nano or qwen3.5-9b. We sample 150 queries as the training set, cap the evolution budget at 20 evaluations, and run the final evaluation on a separate sample of 300 queries.

### A.3.5 ARC-AGI

The optimizer LLM is Opus-4.6 or DeepSeek-v4-pro and the agent LLM is gpt-5.4-nano or qwen3.5-9b. We sample 100 problems from the training set, cap the evolution budget at 20 evaluations, and run the final evaluation on 400 problems sampled from the test set.

## B Implementation Details

### B.1 Algorithm of Outer Propose–Evaluate Loop

The algorithm of outer Propose–Evaluate loop is presented in Algorithm 1.

**Algorithm 1** Outer Propose–Evaluate Loop

**Require:** Proposer $P$, Evaluator $E$, artifact $x$, context $\mathcal{I}$, batch size $k$, verify command VERIFY

$\mathcal{D} \leftarrow \emptyset$ ▷ directions ever proposed
$\mathcal{L} \leftarrow [\,]$ ▷ per-trial analysis log
$m_0 \leftarrow \text{VERIFY}(x)$ ▷ baseline metric
**while not** stopped **do**
  $\mathbf{Q} \leftarrow P(x, \mathcal{D}, \mathcal{L}, \mathcal{I})$
  **for** $p \in \mathbf{Q}$ in ranked order **do**
    $\tilde{x} \leftarrow \text{APPLY}(p, x)$ ▷ in a fresh worktree off main
    $(d, \Delta) \leftarrow E(\tilde{x})$
    **if** $d = \text{KEEP}$ **then**
      $x \leftarrow \tilde{x}$ ▷ fast-forward main
    **end if**
    $\mathcal{L} \leftarrow \mathcal{L} \,\|\, (p, d, \Delta)$
  **end for**
**end while**
**return** $x$

### B.2 Source Code

The source code is available at an anonymous GitHub link:

https://anonymous.4open.science/r/GIOE_08E4/

### B.3 Trial isolation via git worktrees

Each verify trial executes in a dedicated git worktree rather than in the main checkout. For trial $N$ on proposal $k$ the orchestrator creates a fresh worktree branched off main's current `HEAD`,

```
git worktree add -b metajudge/iter-N-prop-k
  .metajudge-worktrees/iter-N-prop-k  HEAD,
```

applies the proposal's edits inside that worktree, commits them there, and lets the Evaluator run VERIFY with the worktree as the working directory. On KEEP the worktree's branch is fast-forwarded into main; on DISCARD the worktree is force-removed and main is unchanged. Cleanup (`git worktree remove -force` followed by `git branch -D`) runs regardless of outcome.

This design enforces the invariant that *main's* `HEAD` *only advances on* KEEP. Three consequences follow. (i) There is no revert step: DISCARD is a pure no-op on the canonical artifact, because the modification was never applied to it. (ii) If the orchestrator crashes mid-trial — during apply, commit, or verify — main is untouched and still reflects the best modifications kept so far; recovery is just a sweep of leftover worktrees and branches whose names match the `metajudge/` prefix. (iii) Because each trial branches off main's *current* `HEAD`, a proposal that was KEPT earlier in a single Inspector queue is already part of the baseline against which subsequent proposals in the same queue are verified; the queue is walked sequentially, with strict-improvement KEEP/DISCARD (Section 3.3) applied against the rolling baseline.

Stale worktrees from a previously interrupted run are swept idempotently at startup by enumerating `git worktree list -porcelain` and removing anything under `.metajudge-worktrees/`, followed by deletion of orphaned `metajudge/` branches. The sweep is safe on a fresh repo and

required after a crash.

### B.4 Cross-iteration state files

The system maintains four state files in the main working directory, all listed in `.gitignore`. They are deliberately kept outside the versioned tree so that the per-trial worktree lifecycle — create, commit in worktree, remove — does not interact with them; the files persist across trials regardless of what happens to any single worktree.

- **`Modification.md`** (append-only). The brainstorm history $\mathcal{D}_t$. Each Proposer call appends one block of five named directions with one-line rationales, including rounds whose proposals were ultimately hard-gated by the Inspector. The file records what has been *considered*, not necessarily what has been *tried*, and exists primarily so the Proposer can avoid re-proposing the same idea by name or mechanism in a later round.

- **`proposal.md`** (overwritten each round). The current batch of five proposals in full detail — files to edit, change summary, rationale, and concrete edits at the symbol-and-line-range level. Consumed by the Inspector for scoring and by the orchestrator for applying edits into the worktree. Unlike the other three files this one carries no history; the round-$t$ batch overwrites the round-$(t-1)$ batch.

- **`Analyzation.md`** (append-only). The mechanism-analysis log $\mathcal{L}_t$. After every trial, the Evaluator appends a block recording the proposal name, the files changed, the prev-best vs. new metric values with their signed difference, the KEEP/DISCARD decision, and a short mechanism analysis: did the change behave as the proposal predicted, what does the outcome imply about the system's bottlenecks, and what direction does it suggest is worth trying next. This is the most-read artifact in the loop: both the Proposer (when brainstorming, §B.5) and the Inspector (when scoring $\mu$ and $\sigma$, Section 3.2) condition on it.

- **`results.tsv`** (append-only). The quantitative trial log: one tab-separated row per verify with columns `iteration`, `timestamp`, `status` (one of `baseline`, `KEEP`, `DISCARD`, `crash`), `metric`, `files_changed`, and a one-line `summary`. The next iteration number is computed as $(\max \texttt{iteration})+1$ at the start of each new trial; it is re-read between trials in the same Inspector queue because a KEEP earlier in the queue will have written a new row.

Two of these files realize the state symbols introduced in Section 3.2: $\mathcal{D}_t$ is the union of direction blocks accumulated in `Modification.md` through round $t$, and $\mathcal{L}_t$ is the sequence of analysis blocks accumulated in `Analyzation.md`. `results.tsv` is the coarser metric-level trace of the same trajectory; the Proposer and Inspector read it for status counts and cross-reference, but the substantive learning signal lives in `Analyzation.md`.

### B.5 Proposer's two-pass brainstorm

The Proposer emits five fresh, concretely-actionable directions per call. No upfront brainstorm is performed at setup: every round generates a new batch conditioned on the running state. Because the Inspector does not cull below the hard gate (Section 3.3), every admissible direction in the batch will be tried, so the batch is constructed to contain five genuinely distinct proposals rather than "one good idea and four fillers."

Brainstorming proceeds in two passes over the state files.

**First pass — mine $\mathcal{L}_t$.** The Proposer reads `Analyzation.md` and extracts: (i) "next-direction" or "this suggests…" hints surfaced in prior Evaluator analyses that have never been acted on, (ii) mechanisms behind previous KEEPs whose effect could plausibly be extended further, and (iii) mechanisms behind DISCARDs whose entire *family* — not just the specific edit — should now be avoided. A direction that has been mentioned across multiple prior analyses but never appears in `results.tsv` is treated as a high-priority candidate: $\mathcal{L}_t$ is read as forward-looking evidence, not merely a record of what happened.

**Second pass — fill out diversity.** After the first pass has contributed its candidates, the remaining slots are used to span different files, layers, and mechanisms. Five minor variants of one idea is an explicitly named failure mode: if recent rounds have all generated proposals from the same family, the Proposer is required to jump to a different layer (data, algorithm, I/O, config, structure) for at least one of the five entries. Family diversity within a single batch translates directly into broader

exploration per outer-loop round, because all five admissible proposals will reach the Evaluator.

**Forbidden shapes.** Three classes of proposal are screened at the Proposer stage rather than left for the Inspector to catch: (a) Goodharting shapes in the sense of Section 3.3 — modifying random seeds, hardcoding metric values, weakening the verify command, leaking the target, conditional measurement-only behavior, or shrinking the workload; (b) direct repeats, by name or by mechanism, of entries already in $\mathcal{D}_t$; (c) vague themes ("refactor for cleanliness") with no concrete measurable mechanism for moving the primary metric. Filtering at the source keeps the Inspector's hard-gate workload reserved for genuinely ambiguous cases and avoids the cost of an all-rejected round when an entire batch would otherwise be screened out.

**Output.** Each round, the Proposer appends a block of five named directions to `Modification.md` and overwrites `proposal.md` with the corresponding five full proposals: files to edit, a two-to-four-sentence change summary, a rationale for why this should move the primary metric, and a bulleted list of concrete edits referencing exact symbols, function names, and (when possible) line ranges or before/after snippets. The orchestrator applies the concrete edits against the worktree without re-deriving intent, so the edit specification must stand alone — the proposal text is the only contract between the Proposer and the worktree apply step.

### B.6 Inspector Scoring Rubrics

The Inspector returns $\sigma(p; \mathcal{H}_t)$ and $\mu(p; \mathcal{L}_t)$ as scalar LLM judgments in $[0, 1]$, each calibrated against a small set of anchor descriptions. We give the rubric prompts below.

**Exploration value $\sigma$ (novelty).** The Inspector reads `results.tsv` and `Analyzation.md` to see what has been tried recently, and scores how *novel* the proposal is against four anchors:

- $\sigma = 0.0$ — essentially a repeat of an already-tried idea;
- $\sigma = 0.3$ — small variant of a recently-tried idea;
- $\sigma = 0.6$ — same family but a different mechanism;
- $\sigma = 1.0$ — entirely new direction / family / file area.

*Streak penalty:* if the last three or more trials have all explored the same family, $\sigma$ for any new proposal in that family is scored below the anchor value above. This is the Inspector-side anti-collapse mechanism of Section 3.2.

**Improvement probability $\mu$.** The Inspector reads `Analyzation.md` for what previous changes have taught about the system, and scores how plausibly the proposal improves the primary metric against three anchors:

- $\mu = 0.0$ — no plausible mechanism, or the analysis log already shows this kind of change does not help;
- $\mu = 0.5$ — a reasonable hypothesis, with no direct evidence either way;
- $\mu = 1.0$ — directly addresses a bottleneck identified in `Analyzation.md`, or extends a successful prior KEEP.

Intermediate values are interpolated between adjacent anchors. The composite ranking sums the two scores with equal weight — $\frac{1}{2}\mu + \frac{1}{2}\sigma$, order-equivalent to the acquisition function $\alpha = \mu + \beta\sigma$ of (1) with $\beta = 1$ on rubrics already normalized to $[0, 1]$.

## C Additional Experimental Results

### C.1 Evolution Curves

Figure 4 shows representative best-reward trajectories during iterative optimization. Compared with one-shot initialization, iterative evolution steadily improves performance over time by accumulating validated changes. The curves also reveal clear differences in exploration behavior: some methods plateau early near their initial solutions, while others continue discovering higher-reward directions throughout the search. In particular, **GIOE** consistently reaches stronger final solutions and maintains sustained improvement over longer horizons.

### C.2 Per-task Results on AutoLab

**Puzzle-and-Challenge Tasks.** Table 5 reports results on six puzzle-and-challenge tasks from AutoLab. **GIOE** achieves the best average performance and obtains the best result on `smallest_game_player` and `discover_sorting`. Overall, the results show that **GIOE** remains competitive across diverse search and optimization-style puzzle tasks.

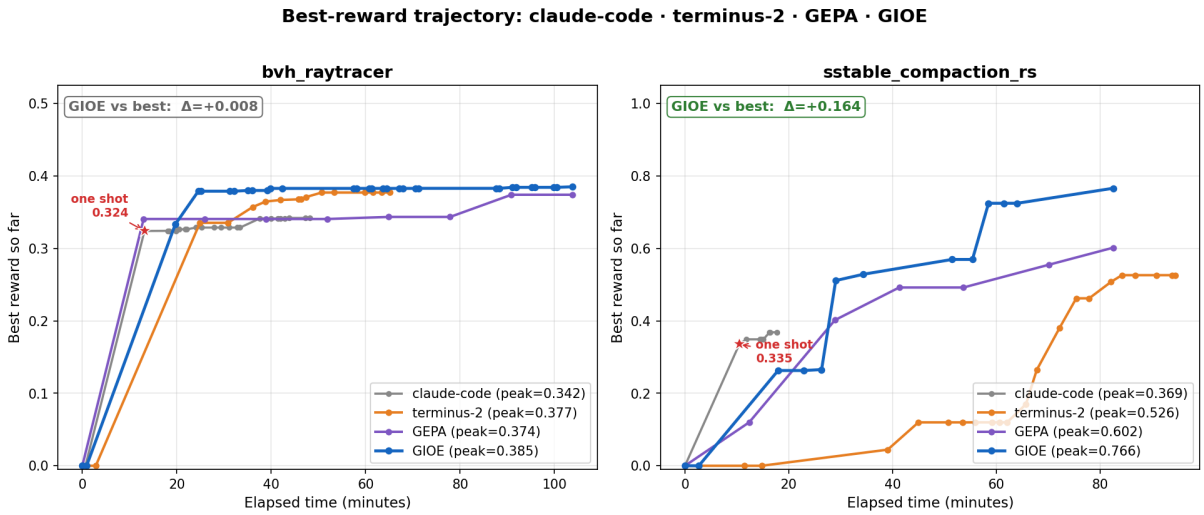


Figure 4: Representative best-reward trajectories during iterative optimization. The plots show the best reward found so far as a function of elapsed optimization time. Compared with one-shot initialization, iterative evolution progressively improves the solution quality through repeated proposal, evaluation, and refinement.

| Method | SMG | FSN | TIO | VS | SGP | DS | Avg. |
|---|---|---|---|---|---|---|---|
| One-shot edit | 0.227 | 0.333 | <u>0.480</u> | 0.000 | 0.761 | 0.467 | 0.378 |
| Terminus-2 | 0.000 | **0.491** | 0.000 | 0.000 | <u>0.850</u> | 0.390 | 0.289 |
| GEPA | **0.946** | **0.491** | 0.433 | 0.000 | 0.636 | 0.584 | 0.515 |
| Claude Code | 0.749 | 0.450 | **0.535** | 0.000 | <u>0.850</u> | <u>0.598</u> | <u>0.530</u> |
| **GIOE** | <u>0.844</u> | <u>0.490</u> | **0.535** | 0.000 | **1.000** | **0.645** | **0.586** |

Table 5: Per-task results on AutoLab puzzle-and-challenge tasks. Bold indicates the best result in each column, and underline indicates the second-best distinct result. Columns abbreviate: SMG = `stack_machine_golf`, FSN = `fredkin_sort_network`, TIO = `toy_isa_opt`, VS = `vliw_scheduler`, SGP = `smallest_game_player`, DS = `discover_sorting`.

**Model Development Tasks.** Table 6 shows results on five model development tasks. **GIOE** achieves the best performance on `data_select_ifeval` and `llm_online_serving`, and ties for the best result on `scaling_law`. It also obtains the highest average score across the model development tasks.

### C.3 Additional Ablation Study

Table 7 reports additional ablations on AutoLab systems-optimization tasks. The results show that the full method achieves the best average score, while removing either Hard-Gated Goodharting Control (HGGC) or Acquisition-Function Scoring (AFS) consistently weakens overall performance.

| Method | MO | SL | DSI | GM | LOS | Avg. |
|---|---|---|---|---|---|---|
| One-shot edit | 0.274 | 0.000 | 0.440 | 0.982 | 1.000 | 0.539 |
| Terminus-2 | 1.240 | **0.997** | 0.827 | **1.071** | 1.004 | 1.028 |
| GEPA | 0.691 | 0.147 | 0.946 | 1.054 | 0.000 | 0.568 |
| Claude Code | **1.755** | 0.000 | 0.858 | **1.071** | 1.017 | 0.940 |
| **GIOE** | 1.186 | **0.997** | **1.017** | 1.054 | **1.059** | **1.063** |

Table 6: Per-task results on AutoLab model development tasks. Bold indicates the best result in each column, and underline indicates the second-best distinct result. Columns abbreviate: MO = `multilingual_ocr`, SL = `scaling_law`, DSI = `data_select_ifeval`, GM = `grpo_multisource`, LOS = `llm_online_serving`.

| Method | FA | BM25 | AES | BVH | KVW | FFT | Blur | Join | Regex | SST | Radix | SHA | Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Ablations** | | | | | | | | | | | | | |
| w/o Brainstorm | 0.411 | 0.454 | 0.637 | 0.353 | 0.316 | 0.502 | 0.362 | 0.568 | 0.490 | 0.493 | 0.498 | **0.091** | 0.431 |
| w/o HGGC | 0.379 | 0.323 | 0.510 | 0.329 | 0.096 | 0.522 | **0.550** | 0.650 | 0.255 | 0.472 | 0.520 | 0.057 | 0.389 |
| w/o AFS | 0.376 | **0.534** | 0.656 | 0.365 | 0.485 | **0.554** | 0.482 | 0.636 | 0.561 | 0.452 | 0.518 | 0.059 | 0.473 |
| w/o Brainstorm+HGGC | 0.395 | 0.402 | 0.695 | **0.398** | 0.600 | 0.545 | 0.498 | 0.668 | 0.442 | 0.378 | 0.522 | 0.069 | 0.468 |
| w/o Brainstorm+AFS | 0.400 | 0.530 | **0.710** | 0.379 | 0.131 | 0.506 | 0.529 | **0.684** | 0.831 | 0.360 | 0.539 | 0.051 | 0.471 |
| w/o HGGC+AFS | 0.408 | 0.415 | 0.701 | 0.374 | 0.510 | 0.522 | 0.464 | 0.657 | 0.524 | 0.519 | 0.502 | 0.080 | 0.473 |
| **Full method** | | | | | | | | | | | | | |
| **GIOE** | **0.433** | 0.442 | 0.701 | 0.385 | **1.000** | 0.541 | 0.492 | 0.659 | **0.925** | **0.766** | **0.582** | **0.412** | **0.612** |

Table 7: Additional ablation results on AutoLab systems-optimization tasks. HGGC denotes Hard-Gated Goodharting Control and AFS denotes Acquisition-Function Scoring. Columns abbreviate `flash_attention`, `bm25_search_go`, `aes128_ctr`, `bvh_raytracer`, `concurrent_kv_wal`, `fft_rust`, `gaussian_blur`, `hash_join`, `regex_engine`, `sstable_compaction_rs`, `radix_sort`, and `sha256_throughput`.