EVOALLOC: A SELF-EVOLVING RESOURCE ALLOCATION AGENT FOR EFFICIENT PROGRAM EVOLUTION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based program evolution relies on evaluation feedback to guide the iterative search for high-performing programs. However, evaluation is often computationally expensive, making it essential to allocate limited resources to candidates that can most effectively advance the search. Existing LLM-based methods typically rely on fixed allocation strategies throughout the search, potentially wasting resources on low-value candidates while overlooking promising ones. We propose **EvoAlloc**, a self-evolving resource-allocation agent that learns from search experience to revise its strategy for allocating computational resources across candidates. EvoAlloc periodically consolidates prior search and allocation outcomes into reusable experience, which informs subsequent strategy revisions. It further uses a counterfactual exploration mechanism to occasionally evaluate candidates denied resources by the allocator, revealing their outcomes to enrich its experience for future strategy updates. Across coding and agent-harness optimization benchmarks, EvoAlloc requires 59–82% fewer full evaluations and 61–89% fewer total LLM tokens to reach baseline-level performance. Moreover, under the same full-evaluation budget, EvoAlloc achieves 8.7–12.0% higher final performance.

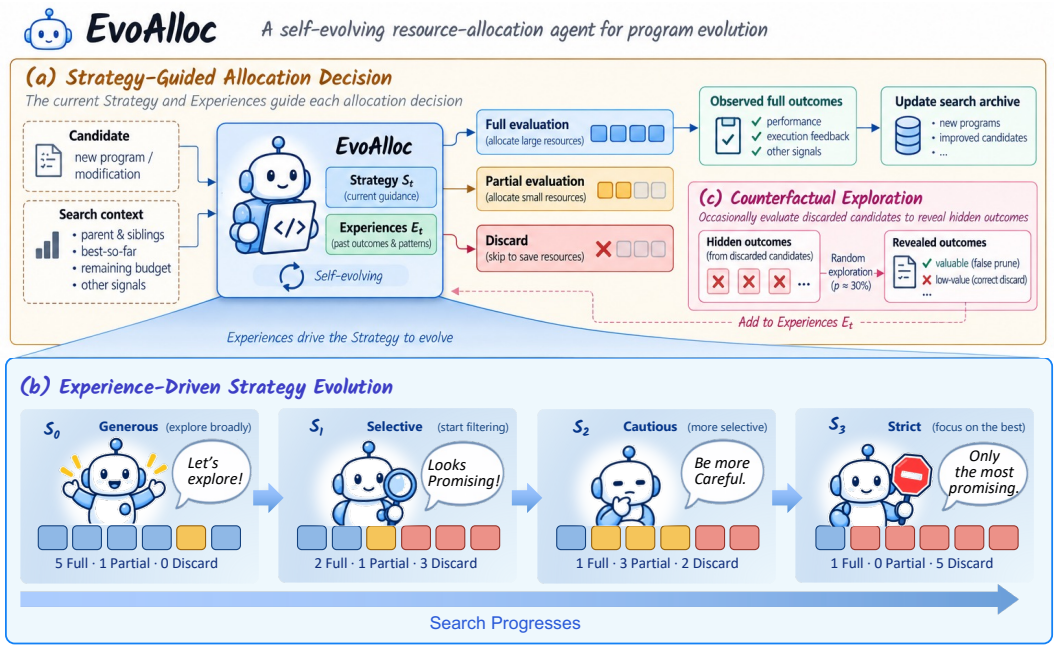


Figure 1: **EvoAlloc, a self-evolving resource allocation agent for program evolution.** (a) Strategy-guided allocation decision, (b) Experience-driven Strategy evolution, and (c) counterfactual exploration jointly adapt how evaluation is allocated as search progresses.

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.

1 INTRODUCTION

Recent advances in LLM-based program evolution have established a general paradigm for discovering and refining executable programs through iterative cycles of generation, evaluation, and selection. By pairing the ability of language models to propose semantically meaningful code modifications with feedback from program execution, these methods have demonstrated strong potential across mathematical and algorithmic discovery (Romera-Paredes et al., 2024), scientific and systems optimization (Novikov et al., 2025), and the automated improvement of coding agents (Lange et al., 2026; Zhang et al., 2026; Wang et al., 2026). Evaluation plays a central role in this process because it directly shapes the trajectory of search. Evaluated programs populate the archive from which future parents are drawn, whereas a promising candidate that goes unevaluated cannot influence what is explored next. At the same time, fully evaluating a candidate can be expensive, requiring execution over many benchmark instances or random seeds, calls to external models or solvers, or multi-step rollouts of LLM agents. Deciding which candidates to evaluate, and how much evaluation each should receive, is therefore not merely a matter of saving compute but a central decision of the search itself.

Existing approaches largely make this decision using an allocation rule specified before search begins. The rule may be a predefined multi-fidelity procedure that promotes candidates through increasing evaluation budgets (Li et al., 2018; 2020), or an LLM that judges, ranks, or filters candidates before full evaluation (Foster et al., 2026). While these methods may condition individual decisions on intermediate results or search history, the allocation strategy itself remains fixed. This leaves valuable information unused. Each evaluated candidate reveals whether the decision to evaluate it was worthwhile, and as search proceeds, these outcomes accumulate into experience about which allocation choices advanced the search. Such experience matters because the value of evaluating a candidate is not a property of the candidate alone. It depends on the evolving state of the search. As the incumbent improves, useful improvements become rarer and the remaining budget shrinks, so what is worth evaluating changes over time. We argue that the allocator should therefore co-evolve with the search, improving its strategy from the outcomes of its own decisions.

Building on this view, we introduce EvoAlloc, a self-evolving allocation agent embedded in the program-evolution loop (Figure 1). EvoAlloc maintains an explicit allocation strategy that determines whether and how much to evaluate each candidate. Rather than keeping this strategy fixed, EvoAlloc evolves it as the search proceeds, drawing on a reflective memory that distills the search history and past allocation outcomes into reusable experience. Proposed revisions are adopted when shared validation favors them, prioritizing fewer missed new-best candidates and then lower evaluation cost. This self-evolution loop, however, creates a selective-feedback challenge: the agent's own decisions determine which outcomes become observable. EvoAlloc therefore incorporates counterfactual exploration to gather additional experience beyond what its current strategy would otherwise generate. Across coding and agent-harness optimization benchmarks, EvoAlloc improves final performance by 8.7–12.0% over baseline under the same evaluation budget, while requiring 59–82% fewer full evaluations and 61–89% fewer recorded LLM tokens to match the baseline's final performance. Ablations confirm the contributions of strategy evolution and counterfactual exploration.

Our main contributions are:

1. **We introduce a view of evaluation allocation as an evolving component of program evolution.** We reframe allocation from fixed per-proposal judgments into an evolving decision process that uses the outcomes of prior allocation decisions to inform future ones.
2. **We introduce EvoAlloc, a self-evolving resource-allocation agent.** EvoAlloc acts through an explicit strategy, accumulates reusable experience from its interaction with the search, explores counterfactual outcomes, and revises future allocation behavior.
3. **We provide an empirical analysis of how evolving evaluation allocation reshapes program search.** Across coding and agent-harness benchmarks, we show that adaptive allocation improves search under limited budgets and analyze how it changes resource use, candidate coverage, and the resulting search trajectories.

2 Related Work

Program Evolution and Self-Improving Search. Program evolution has a long history in genetic programming, where evolutionary variation and selection are applied directly to executable programs (Cramer, 1985; Dickmanns et al., 1987). LLM-based systems extend this paradigm by using language models to generate, revise, and recombine candidate programs within iterative search (Romera-Paredes et al., 2024; Novikov et al., 2025). Recent work has further made the search process itself more adaptive through mechanisms such as parent selection, novelty control, population management, and learned search strategies (Lange et al., 2026; Assumpção et al., 2025; Wan et al., 2025). A related line of work considers self-improvement beyond the candidate programs themselves. Early meta-evolution explored evolving program-modification mechanisms (Schmidhuber, 1987), while the Gödel Machine formalized self-improvement as rewriting its own code once a modification is proven useful (Schmidhuber, 2007). More recent systems such as Darwin Gödel Machine and Huxley-Gödel Machine pursue this idea empirically by evolving coding agents and their problem-solving procedures (Zhang et al., 2026; Wang et al., 2026). While these approaches largely adapt how candidates are generated and searched, the evaluation mechanism that allocates compute typically remains fixed; EvoAlloc allows this mechanism to evolve as well, drawing on accumulated search experience.

Selective Evaluation and Adaptive Resource Allocation. Evaluation allocation has been studied primarily in settings where full evaluation is expensive, particularly in black-box optimization such as hyperparameter tuning. Classical approaches exploit inexpensive signals before committing to full evaluation: surrogate-assisted methods use predictive models as proxies for candidate quality (Snoek et al., 2012; Jin, 2011), while multi-fidelity methods acquire low-cost partial evaluations and progressively devote more resources to promising candidates (Li et al., 2018; 2020). In algorithm configuration, racing and adaptive capping similarly decide how much evaluation each configuration receives, terminating runs or discarding candidates once they are unlikely to outperform the best found so far (Hutter et al., 2009; López-Ibáñez et al., 2016). Similar heuristics have become common in LLM-based program evolution, including cascade evaluation that progressively filters candidates through increasingly comprehensive tests (Novikov et al., 2025) and novelty-based rejection of near-duplicate proposals (Lange et al., 2026). Beyond such heuristics, LLMs have also been used to rank or filter candidates before costly evaluation, through preference-based and agentic selection mechanisms (Foster et al., 2026). These approaches can adapt allocation decisions to observed outcomes while retaining predefined allocation rules. EvoAlloc instead maintains an explicit allocation strategy and revises its criteria using accumulated search and evaluation outcomes.

3 Problem Formulation

3.1 Evaluation Allocation in Program Evolution

Consider a program-evolution process that iteratively generates candidate programs, evaluates them, and uses the evaluated programs to guide subsequent generation. We formulate evaluation allocation within this process as a sequential decision problem solved by an allocation agent $\mathcal{A}$, which distributes a limited evaluation budget across candidates to maximize the best program score reached by the search. At each search step t, a candidate program x_t is generated, and the agent observes a decision-time context c_t containing information available before evaluating x_t, such as the parent program and its score, the best score found so far, and the remaining budget.

For candidate x_t, allocation may involve multiple decisions. Let $j = 0, 1, \ldots, K_t$ index the decisions within its allocation episode, and let $a_{t,j}$ denote the allocation action taken at decision j. Each intermediate action $a_{t,j}$ ($j < K_t$) acquires evaluation evidence $e_{t,j}$ about x_t, which becomes available to subsequent decisions, and the final action a_{t,K_t} either commits the candidate to full evaluation or stops further evaluation. The agent may condition each decision on the candidate, the decision-time context, and the evidence acquired earlier in the same episode:

$$a_{t,j} = \mathcal{A}(x_t, c_t, e_{t,<j}), \tag{1}$$

where $e_{t,<j} := (e_{t,0}, \ldots, e_{t,j-1})$, with $e_{t,<0} := \varnothing$. The resulting allocation episode is $\tau_t = (a_{t,0}, e_{t,0}, \ldots, a_{t,K_t-1}, e_{t,K_t-1}, a_{t,K_t})$. Let $y_t \in \mathbb{R}$ denote the full-evaluation score of x_t, which is defined regardless of whether x_t is actually fully evaluated. The score observed from the allocation

episode is

$$y_t^{\text{obs}} = \begin{cases} y_t, & \text{if } x_t \text{ is fully evaluated,} \\ \varnothing, & \text{otherwise.} \end{cases}$$

Only y_t^{obs} is available to the search and to the agent; the scores of candidates that are not fully evaluated remain hidden.

Let b_t denote the cumulative resource consumed after proposal t, measured, for example, by completed full evaluations or cumulative LLM tokens. Using this resource variable, we evaluate $\mathcal{A}$ through the resulting search trajectory. Let Q_0 denote the best score among the initial programs. For any resource level b, define

$$Q^\star_{\mathcal{A}}(b) = \max\left(\{Q_0\} \cup \left\{y_t^{\text{obs}} : t \in \mathbb{N},\ b_t \le b,\ y_t^{\text{obs}} \neq \varnothing\right\}\right), \tag{2}$$

which gives the best fully evaluated program score obtained after consuming at most resource b. Given a total resource budget B, the objective is to choose an allocation agent $\mathcal{A}$ that maximizes $\mathbb{E}[Q^\star_{\mathcal{A}}(B)]$, where the expectation is over the stochasticity of both the search process and the allocation agent. Conversely, for a target score q, the resource required to reach it is $C_q(\mathcal{A}) = \inf\{b \ge 0 : Q^\star_{\mathcal{A}}(b) \ge q\}$. These objectives make evaluation allocation a search-level sequential decision problem rather than independent proposal-level decisions. Resources saved on one candidate remain available for later search, while fully evaluated candidates can influence which programs are explored next.

3.2 Self-Evolving Allocation from Search Experience

Across proposals, the agent accumulates the interaction history $\mathcal{H}_t = \{(x_i, c_i, \tau_i, y_i^{\text{obs}})\}_{i<t}$, which contains all allocation interactions before search step t. Here, τ_i records the allocation actions and intermediate evidence for candidate x_i. Episodes without Full evaluation are also included, with $y_i^{\text{obs}} = \varnothing$, although these candidates cannot participate in subsequent program evolution.

We represent the allocation agent over the search as $\mathcal{A} = (\mathcal{A}_t)_{t\ge 0}$, where $\mathcal{A}_t$ denotes its decision mechanism at search step t. A fixed allocation agent may make different decisions as the proposal, search context, and within-episode evidence change, while its decision mechanism remains unchanged: $\mathcal{A}_t = \mathcal{A}_0$ for all t. We call an allocation agent *self-evolving* when it can also revise this mechanism using the accumulated interaction history: $\mathcal{A}_{t+1} = \mathcal{U}(\mathcal{A}_t, \mathcal{H}_{t+1})$, where $\mathcal{U}$ denotes the agent-update process and leaves $\mathcal{A}_t$ unchanged when no update is performed. Section 4 instantiates this formulation in EvoAlloc, where $\mathcal{A}_t$ is guided by an explicit allocation Strategy S_t and reusable Experiences E_t distilled from observed search outcomes.

4 Method

EvoAlloc. EvoAlloc is a self-evolving evaluation-allocation agent for program evolution. It follows an explicit strategy that allocates evaluation resources to each candidate using the information available at decision time. As outcomes accumulate, EvoAlloc distills prior allocation episodes into reusable experience and uses it to evolve the strategy. It further uses counterfactual exploration to reveal outcomes of selected candidates that would otherwise remain unevaluated, providing additional feedback for subsequent strategy evolution. The following sections describe these three mechanisms: explicit evaluation allocation (Section 4.1), experience-driven strategy evolution (Section 4.2), and counterfactual exploration (Section 4.3).

4.1 Explicit Evaluation Allocation Strategy

EvoAlloc maintains an explicit natural-language allocation strategy S_t that persists across candidates and is revised as the search progresses (Sec. 4.2). Rather than making each allocation decision from its current inputs alone, S_t captures search-level guidance for allocating evaluation resources. It specifies both the allocator's overall selectivity and conditional guidance for choosing allocation actions under different search contexts. Representative strategy trajectories are provided in Appendix C.

In EvoAlloc, the agent $\mathcal{A}_t$ at search step t is determined by its current Strategy S_t and Experiences E_t, which are provided to a fixed allocator LLM together with the inputs of Eq. (1). Here, c_t captures the decision-time search context, including the parent score, evaluated sibling outcomes, current best score, and remaining budget, while $e_{t,<j}$ contains evidence acquired earlier in the same allocation

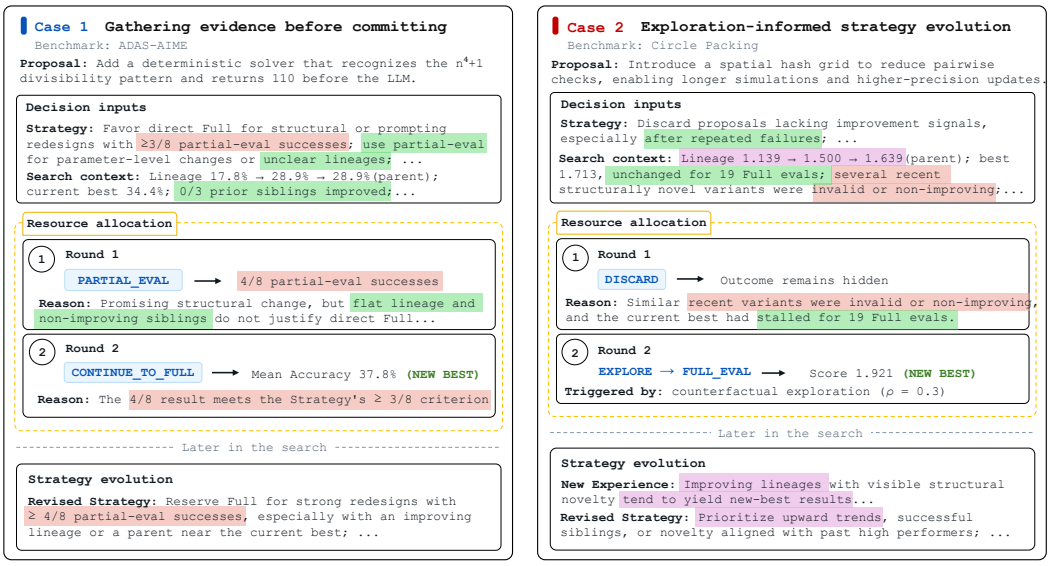


Figure 2: **Information flow between the strategy and allocation decisions in EvoAlloc.** Each case reads top to bottom: the strategy and search context inform the allocation agent's decisions, whose outcomes feed into later strategy updates. Proposals describe changes to the parent program. Matching highlights link related information across stages. Content is shortened for clarity. Strategy updates reflect feedback accumulated over multiple proposals, not the illustrated proposal alone.

episode. Across candidates, E_t summarizes reusable evidence from observed outcomes, and S_t provides the current run-level allocation guidance. The allocation action space is task-dependent. In all settings, EvoAlloc may choose `FULL_EVAL` to commit the candidate to full evaluation or `DISCARD` to stop further evaluation. When a meaningful lower-cost intermediate evaluator is available, the agent may additionally choose `PARTIAL_EVAL` to acquire further evidence. After such evidence is observed, the subsequent action is either `CONTINUE_TO_FULL` or `STOP`. These actions instantiate the multi-step allocation episode defined in Section 3.1.

4.2 Experience-Driven Strategy Evolution

EvoAlloc implements the agent-update process $\mathcal{U}$ of Section 3.2 by updating two complementary components from observed outcomes. Experiences E_t summarize recurring patterns across observed cases, and the Strategy S_t translates these patterns into allocation criteria.

Experience updates. After every k_E newly observed Full outcomes, the LLM consolidates the new cases into E_t by adding, updating, or removing Experiences. Each Experience maintains the indices of its supporting and contradicting cases and a status of *tentative*, *active*, or *stale*. The case indices are chronological, preserving how the evidence for an Experience changes over time. New patterns enter as *tentative*; during later updates, the LLM may promote well-supported Experiences to *active* or mark outdated ones as *stale* as new evidence accumulates. Stale Experiences remain as historical records but no longer guide allocation.

Strategy evolution. EvoAlloc starts from an initial strategy S_0 derived from a few example cases. Each run begins with a short warm-up in which the allocator acts under S_0 but every candidate is fully evaluated, seeding run-specific evidence. Thereafter, every k_S newly observed Full outcomes, EvoAlloc reflects on the current strategy, Experiences, and recent cases to propose a challenger $\widetilde{S}$. The challenger is validated against the current strategy on subsequent candidates that induce different allocation paths under the two strategies (disagreements). For each disagreement, EvoAlloc executes the union of both strategies' requested evaluations and shares the resulting evidence. Validation ends after two disagreements with an observed Full outcome. Sharing evaluations prevents outcomes requested only by the challenger from being censored in favor of the incumbent (Appendix D.7). The two strategies are then compared lexicographically on the realized consequences of their decisions:

fewer missed new-best candidates, then fewer Full evaluations, then fewer Partial evaluations. The challenger is adopted if it is better on the first criterion where the two differ; otherwise, the current strategy is retained. This grounds strategy updates in observed allocation consequences rather than in LLM judgment alone. Implementation details and prompts are provided in Appendix E.

4.3 Counterfactual Exploration

Strategy evolution requires evaluation feedback, yet selective allocation itself determines which feedback becomes observable. As defined in Sec. 3.1, candidates that terminate without full evaluation have $y_t^{\text{obs}} = \varnothing$. The resulting feedback is therefore filtered by the current strategy: systematic allocation errors can remain hidden because the corresponding candidates are never evaluated. To expose such missing feedback, EvoAlloc introduces *counterfactual exploration*. For each candidate whose allocation episode terminates in `DISCARD` or `STOP`, EvoAlloc independently samples $z_t \sim \text{Bernoulli}(\rho)$, where ρ is the exploration rate (0.3 by default). When $z_t = 1$, EvoAlloc executes `EXPLORE_EVAL`, fully evaluating the otherwise skipped candidate and revealing $y_t^{\text{obs}} = y_t$. Exploratory outcomes are marked separately from strategy-selected feedback. Strong outcomes provide counterevidence to the current allocation criteria, while weak ones support them; both inform subsequent experience and strategy updates. Once fully evaluated, an explored candidate participates in the underlying program search as usual.

Together, these components form a closed loop: the Strategy guides the agent's decisions, and the outcomes of these decisions are used to revise the Strategy. Figure 2 illustrates this loop through two cases. In Case 1, the Strategy's evidence threshold leads the agent to request Partial evaluation before committing to Full evaluation, and accumulated outcomes later refine this threshold. In Case 2, counterfactual exploration reveals the outcome of a candidate the agent had discarded, exposing feedback that would otherwise remain unobserved and informing a revised Strategy.

5 Experiments

5.1 Experimental Setup

Implementation and protocol. We implement EvoAlloc within the ShinkaEvolve program-evolution framework (Lange et al., 2026), with all methods sharing the search pipeline except for evaluation allocation. Unless otherwise stated, experiments use five independent seeds. We report mean best-so-far performance against Full evaluations and cumulative LLM tokens, together with final best performance. Our primary efficiency metric is matched-target cost. We first average best-so-far scores and cumulative token usage across five seeds at each completed Full-evaluation index. We then find the first index at which the mean score reaches a common target and report that index and its corresponding mean cumulative token usage (Appendix D.3).

Benchmarks. We evaluate EvoAlloc across program-evolution settings spanning reasoning-harness optimization and solution-program optimization, with differing access to intermediate evaluation feedback. **ADAS-AIME** evolves programmatic reasoning harnesses around a fixed Qwen3-8B solver on the 30 AIME 2024 problems. Each candidate is scored by its mean accuracy over three independent runs. A Full evaluation uses all 30 problems, while a Partial evaluation uses the first eight before deciding whether to continue to Full. **Circle Packing** evolves Python programs that arrange 26 non-overlapping circles within the unit square to maximize the sum of their radii. Its evaluator is deterministic, and since no lower-cost intermediate evaluation is available, the allocator chooses directly between Full and Discard.

Baselines. We compare against four baselines within the same search framework. **ShinkaEvolve** (Lange et al., 2026) fully evaluates every proposal that passes the standard checks, providing a reference without evaluation allocation. **One-Step LLM Allocator** uses the same allocator LLM and search context as EvoAlloc, but makes each allocation decision directly without an explicit Strategy or accumulated Experience. **Lineage-UCB** is an LLM-free adaptive allocator that builds a Beta posterior from parent-improvement outcomes in the local lineage and applies a fixed UCB rule for evaluation allocation. **RPM** (Foster et al., 2026) is a recent LLM-based evaluation-allocation method that generates 15 proposals at each selection step and uses pairwise LLM comparisons in

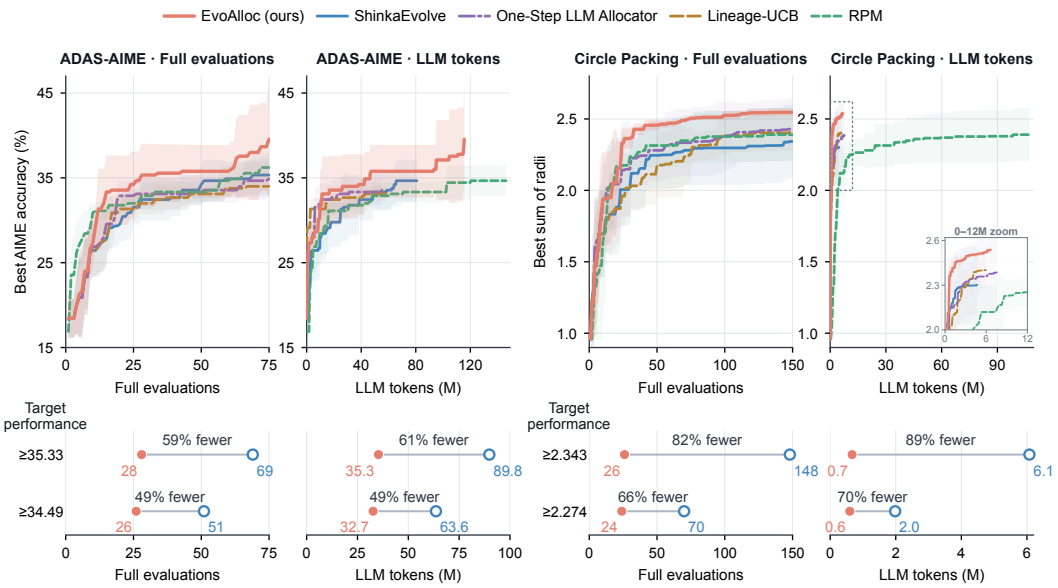


Figure 3: **Main results on ADAS-AIME and Circle Packing.** Curves show five-seed mean best-so-far performance (± 1 std.) against Full evaluations and cumulative LLM tokens. Bottom panels report the resources required to reach ShinkaEvolve's final-performance and 95%-gain targets. Token costs are the mean cumulative usage at the first target-reaching Full-evaluation index of the evaluation-aligned mean performance curve.

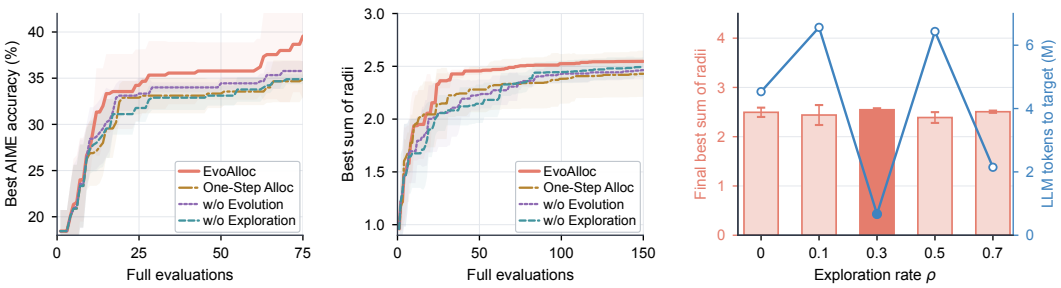
a knockout tournament to select one for Full evaluation. Implementation details, hyperparameters, token accounting, and baseline adaptations are provided in Appendices B and D.

5.2 Main Results

Figure 3 shows that EvoAlloc achieves the highest mean final performance among all evaluated methods on both benchmarks. Relative to ShinkaEvolve, the underlying program-evolution framework, EvoAlloc improves mean final performance by 12.0% on ADAS-AIME and 8.7% on Circle Packing. It also substantially reduces matched-target cost, requiring 59–82% fewer Full evaluations and 61–89% fewer LLM tokens to reach ShinkaEvolve's final-performance target. LLM-token cost includes all input and output tokens consumed during search, including proposal generation, benchmark evaluation, and method-specific allocation or selection calls. The improvements persist across both evaluation settings: on Circle Packing, where no Partial evaluation is available, EvoAlloc still benefits from selectively allocating Full evaluations, showing that the gains are not simply due to access to an additional evaluation stage. The consistent advantage over the One-Step LLM Allocator further supports the value of maintaining an explicit Strategy and accumulated Experience across the search rather than making independent LLM allocation decisions. Complete statistics, matched-target costs, and auxiliary trajectory AUC are reported in Appendix D.

5.3 Ablation and Sensitivity Analysis

Figure 4(a–b) compares variants of EvoAlloc. **One-Step Alloc** replaces the maintained Strategy with independent one-step allocation decisions, using the same allocator LLM and decision-time search context as EvoAlloc. **w/o Evolution** uses fixed custom allocation guidance, distilled from baseline run examples and manually refined (Appendix D.1). **w/o Exploration** removes counterfactual exploration while leaving the remaining adaptation mechanisms unchanged. Across both benchmarks, the full EvoAlloc achieves the strongest mean best-so-far trajectories. The full EvoAlloc outperforms both one-step allocation and fixed custom guidance. Removing exploration also weakens performance, reflecting the value of counterfactual outcomes that selective allocation would otherwise leave unobserved. Figure 4(c) varies the exploration rate ρ on Circle Packing. Final performance is similar across rates, but the LLM tokens needed to reach ShinkaEvolve's final performance vary



(a) ADAS-AIME (b) Circle Packing (c) Exploration rate sensitivity

Figure 4: **Component ablations and exploration-rate sensitivity.** (a–b) Five-seed online ablations under matched Full-evaluation budgets; shading denotes ± 1 standard deviation. (c) Exploration-rate sweep on Circle Packing over $\rho \in \{0, 0.1, 0.3, 0.5, 0.7\}$; bars show the final best sum of radii, and the line shows the LLM tokens required to reach ShinkaEvolve's final performance.

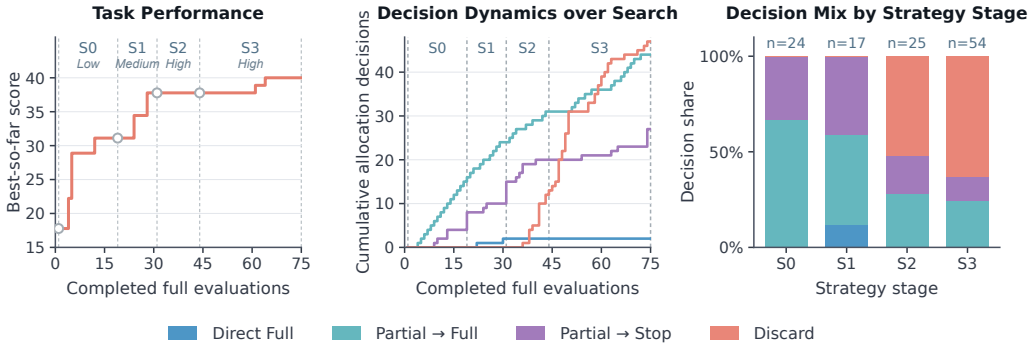


Figure 5: **Evolution of allocation behavior on ADAS-AIME.** A representative 75-full-evaluation run showing best-so-far performance and allocation decisions across S0–S3. Allocation panels show decisions before counterfactual exploration, with explored proposals counted by their original allocation decision. Dashed lines mark the adoption of validated Strategies.

substantially, with $\rho = 0.3$ the most efficient. Too little exploration provides limited feedback on skipped candidates, whereas too much offsets the savings from selective allocation.

5.4 How the Allocation Strategy Evolves and Reshapes Search

A good allocation strategy changes as search progresses. What is worth evaluating depends on the state of the search: early outcomes are hard to predict and gains are frequent, so broad evaluation is informative, whereas later gains become rare and the budget is better spent on fewer, stronger candidates. EvoAlloc's evolved strategies follow this shift (Figure 5; see Appendix C.2 for Circle Packing). On ADAS-AIME, allocation moves from gathering evidence through Partial evaluation with no direct discards to discarding most proposals outright. This greater selectivity does not stop discovery, as EvoAlloc still finds a late improvement. These stage-wise changes support revising allocation criteria as search experience accumulates.

Savings become exploration. Selective allocation does more than reduce cost: every evaluation withheld from a weak candidate is reinvested in further search. With the same Full-evaluation budget on the same seed, EvoAlloc examines 124 proposals on ADAS-AIME versus 75 for ShinkaEvolve (Figure 6), and 302 versus 150 on Circle Packing (Appendix C.3). The extra proposals expand the tree with additional siblings and descendants. The candidates EvoAlloc chooses to evaluate also tend to score higher, even though each choice is made before the Full outcome is known, so the broader coverage does not come at the expense of evaluation quality. This reinvestment is why EvoAlloc improves final performance under a fixed budget, not only the cost of reaching a given target.

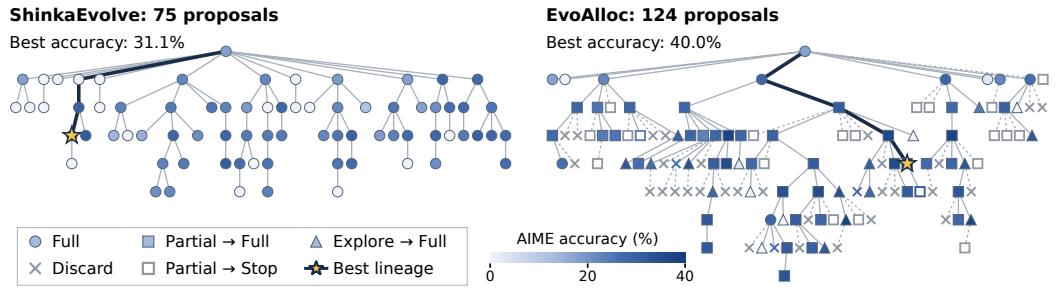


Figure 6: **ADAS-AIME search trees.** Both methods complete 75 full evaluations with the same seed. Shapes encode allocation decisions; colors indicate observed Full-evaluation accuracy. Gray markers denote proposals without Full outcomes; stars mark the best programs, and highlighted edges trace their lineages.



Figure 7: **Allocation decisions under ablations on ADAS-AIME.** Five-seed mean cumulative proposal counts by allocation decision as Full evaluations accumulate; the stacked total gives the number of proposals considered.

EvoAlloc components jointly shape allocation behavior. Figure 7 shows how removing each component changes allocation behavior on ADAS-AIME. One-Step Alloc sees the same search history, but without a Strategy that consolidates it into explicit criteria, the LLM cannot confidently decline, and it therefore sends most proposals to Full evaluation and saves little budget. With fixed custom guidance, allocation remains Partial-heavy throughout the search, unlike the stage-wise shifts of EvoAlloc. Without exploration, discarded outcomes stay hidden, leaving no counterexamples to the current criteria, so pruning errors go uncorrected and falsely pruned candidates never enter the archive as parents. Effective allocation thus requires explicit criteria, their evolution with the search, and exploration to check them against hidden outcomes.

6 Conclusion and Limitations

We introduced EvoAlloc, a self-evolving agent that treats evaluation allocation as an evolving component of program evolution. EvoAlloc revises an explicit allocation Strategy from accumulated Experiences and uses counterfactual exploration to recover feedback that selective allocation would otherwise leave hidden. Across ADAS-AIME and Circle Packing, EvoAlloc improves final performance and reaches the reference final-performance targets with 59–82% fewer Full evaluations and 61–89% fewer LLM tokens. These findings point toward self-improving systems that learn not only what to search, but also where to spend their evaluation effort.

Our study examines self-evolving allocation at the evaluation layer on two benchmarks with contrasting evaluators, within a single search run and with generation, evaluators, and update mechanisms held fixed. Extending resource allocation to generation and allowing evaluation and update mechanisms to evolve could broaden the scope of self-improvement (Robeyns et al., 2025; Zhang et al., 2026). Retaining allocation experience across runs could allow future allocators to build on previously learned strategies, supporting longer-term knowledge accumulation and inheritance, as learngenes do by passing condensed knowledge across generations of agents (Feng et al., 2025). More broadly, adaptive resource allocation could help systems pursuing open-ended self-improvement continually refine how they invest finite computation in future progress.

AI USE STATEMENT

Generative AI tools were used to assist with code implementation and debugging, as well as language editing and presentation of the manuscript. The core research idea and experimental design were developed by the authors. AI-assisted code was reviewed and tested by the authors before use in the experiments, and all AI-assisted manuscript content was manually reviewed and revised for technical correctness and consistency with the reported results. The authors take full responsibility for the final content of the paper.

REPRODUCIBILITY STATEMENT

We provide detailed information to support reproducibility in the appendix, including the EvoAlloc pseudocode, implementation details and hyperparameters, benchmark and evaluation protocols, budget accounting, and all prompts required to reproduce the method. We additionally report five-seed results and allocation breakdowns to facilitate verification of the main findings. The code and configurations will be released upon acceptance.

REFERENCES

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

Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, pp. 183–187, 1985. URL https://dl.acm.org/doi/10.5555/645511.657085.

Dirk Dickmanns, Jürgen Schmidhuber, and Andreas Winklhofer. Der genetische algorithmus: Eine implementierung in prolog. Fortgeschrittenenpraktikum, Institut für Informatik, Lehrstuhl Prof. Radig, Technische Universität München, 1987. URL https://people.idsia.ch/~juergen/Fopra_DD_JS_AW_searchable.pdf.

Fu Feng, Jing Wang, Xu Yang, and Xin Geng. Learngene: Inheritable "genes" in intelligent agents. *Artificial Intelligence*, 348:104421, 2025. doi: 10.1016/j.artint.2025.104421.

Thomas Simon Foster, Bassel Al Omari, Tingchen Fu, Thomas Mann, Carl Domond, Lucia Cipolina-Kun, Bhavul Gauri, Muna Aghamelu, Alexander D. Goldie, Eryk Helenowski, Jean-Christophe Gagnon-Audet, Alberto Pepe, Saba Nazir, Daniel Izcovich, Noam Levi, Rishi Hazra, Karen Hambardzumyan, Nicolas Baldwin, Xian Li, Martin Josifoski, Paris Giampouras, Masoud Jalili Sabet, Anya Sims, Hela Momand, Tatiana Shavrina, Despoina Magka, Jason Weston, Yulin Wang, Anirudh Goyal, João Henriques, Yoram Bachrach, Emily McMilin, and Jakob Nicolaus Foerster. AI research preference models. *arXiv preprint arXiv:2608.13940*, 2026. URL https://arxiv.org/abs/2608.13940.

Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

Yaochu Jin. Surrogate-assisted evolutionary computation: Recent advances and future challenges. *Swarm and Evolutionary Computation*, 1(2):61–70, 2011.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/7886b9bafe76c52fd568db10ff9772df-Abstract-Conference.html.

Liam Li, Kevin Jamieson, Afshin Rostamizadeh, Ekaterina Gonina, Jonathan Ben-tzur, Moritz Hardt, Benjamin Recht, and Ameet Talwalkar. A system for massively parallel hyperparameter tuning. In *Proceedings of Machine Learning and Systems*, volume 2, pp. 230–246, 2020.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018.

Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Thomas Stützle, and Mauro Birattari. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016. doi: 10.1016/j.orp.2016.09.002.

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.

Maxime Robeyns, Martin Szummer, and Laurence Aitchison. A self-improving coding agent. *arXiv preprint arXiv:2504.15228*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. doi: 10.1038/s41586-023-06924-6.

Jürgen Schmidhuber. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook. Diploma thesis, Technische Universität München, 1987.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In Ben Goertzel and Cassio Pennachin (eds.), *Artificial General Intelligence*, pp. 199–226. Springer, 2007. Also arXiv:cs/0309048 (2003).

Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in Neural Information Processing Systems*, volume 25, pp. 2951–2959, 2012.

Chunhui Wan, Xunan Dai, Zhuo Wang, Minglei Li, Yanpeng Wang, Yinan Mao, Yu Lan, and Zhiwen Xiao. LoongFlow: Directed evolutionary search via a cognitive plan-execute-summarize paradigm. *arXiv preprint arXiv:2512.24077*, 2025. URL https://arxiv.org/abs/2512.24077.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations (ICLR)*, 2026.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Tjarko Lange, and Jeff Clune. Darwin gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations (ICLR)*, 2026.

APPENDIX

A EVOALLOC ALGORITHM

Algorithm 1 highlights EvoAlloc-specific operations. Parent selection, proposal generation, archive updates, and benchmark evaluation follow the underlying search procedure.

Algorithm 1: Program evolution with EvoAlloc

Input: *initial programs*; *example cases*; *Full-evaluation budget* B; *exploration rate* ρ; *update intervals* k_E, k_S.
Output: *best program* x^*

$\mathcal{P} \leftarrow$ INITARCHIVE(initial programs); $H \leftarrow \emptyset$; $n \leftarrow 0$ ▷ archive and interaction history
$S \leftarrow S_0 \leftarrow$ INITSTRATEGY(example cases); $E \leftarrow \emptyset$; $\tilde{S} \leftarrow \emptyset$
while $n < B$ **do**
  $x_p \leftarrow$ SELECTPARENT($\mathcal{P}$); $x \leftarrow$ GENERATEPROPOSAL($x_p, \mathcal{P}$)
  $c \leftarrow$ BUILDCONTEXT($x_p, \mathcal{P}, H, n, B$) ▷ decision-time context
  $\tau \leftarrow$ ALLOCATE($x, c; S, E$) ▷ executes and caches any Partial evaluation
  $\tilde{\tau} \leftarrow$ ALLOCATE($x, c; \tilde{S}, E$) **if** $\tilde{S} \neq \emptyset$ **else** $\emptyset$
    ▷ shadow allocation; reuses cached Partial evidence
  if τ ends in Full **then**
    $y^{\text{obs}} \leftarrow$ FULLEVAL(x)
  else if not WARMUPCOMPLETE(H) **then**
    $y^{\text{obs}} \leftarrow$ FULLEVAL(x)
  else if $\tilde{\tau}$ ends in Full **then**
    $y^{\text{obs}} \leftarrow$ FULLEVAL(x) ▷ shared validation (Sec. 4.2)
  else
    sample $z \sim \text{Bernoulli}(\rho)$
    if $z = 1$ **then** $y^{\text{obs}} \leftarrow$ FULLEVAL(x)
    else $y^{\text{obs}} \leftarrow \emptyset$
  $H \leftarrow H \cup \{(x, c, \tau, \tilde{\tau}, y^{\text{obs}})\}$
  if $y^{\text{obs}} \neq \emptyset$ **then** $n \leftarrow n + 1$; $\mathcal{P} \leftarrow$ UPDATEARCHIVE($\mathcal{P}, x, y^{\text{obs}}$)
  if k_E new Full outcomes since the last Experience update **then**
    $E \leftarrow$ UPDATEEXPERIENCES(E, H)
  if $\tilde{S} \neq \emptyset$ **and** VALIDATIONCOMPLETE(H) **then**
    $S \leftarrow$ VALIDATESTRATEGY($S, \tilde{S}, H$); $\tilde{S} \leftarrow \emptyset$ ▷ adopt or retain
  else if $\tilde{S} = \emptyset$ **and** WARMUPCOMPLETE(H) **and**
    k_S new Full outcomes since the last reflection **then**
    $\tilde{S} \leftarrow$ REFLECTSTRATEGY(S, E, H) ▷ one challenger at a time
return BEST($\mathcal{P}$)

Here, $\mathcal{P}$ is the search archive and H the interaction history (Section 3.2), retaining original allocation episodes even when outcomes are later revealed. ALLOCATE follows the allocation procedure in Section 4.1. During validation, the union of requested evaluations is executed once and the resulting evidence is shared. All Full evaluations, including warm-up, validation, and exploration, count toward B.

B Implementation and Benchmarks

B.1 Models and EvoAlloc configuration

EvoAlloc is implemented on top of the public ShinkaEvolve program-evolution loop (Lange et al., 2026). Unless otherwise stated, substrate hyperparameters follow the default ShinkaEvolve configuration. All language models are open-weight and locally served on NVIDIA A100 GPUs, with the same serving setup used across all compared methods. Table 1 summarizes the backbone models, the main EvoAlloc configuration, and the benchmark-specific settings.

Table 1: **Implementation and benchmark configuration.** Shared model and controller settings with benchmark-specific budgets, action spaces, and Partial-evaluation availability.

Backbone and Models		
Proposer LLMs	GLM-4.5-Air; gpt-oss-120b; Qwen3.6-35B-A3B	
ShinkaEvolve meta model	Qwen3.6-35B-A3B	
ShinkaEvolve novelty model	Qwen3.6-35B-A3B	
Embedding model	Qwen3-Embedding-0.6B	
Proposer temperatures	$[0.0,\ 0.5,\ 1.0]$	
Max tokens	16,384	
EvoAlloc Configuration		
Allocator and self-evolution LLM	Qwen3-Coder-Next	
Adaptive warm-up	Requires ≥ 1 new-best and ≥ 1 non-new-best outcome	
Experience update interval k_E	5 new Full outcomes	
Strategy update interval k_S	10 new Full outcomes	
Strategy validation window	2 allocation disagreements with known Full outcomes	
Experience capacity	4 active/tentative + 4 stale	
Counterfactual exploration rate	0.30	
Benchmark-Specific Configuration		
	Circle Packing	**ADAS-AIME**
Objective / metric	Packing score	AIME accuracy
Evaluation model	Deterministic evaluator	Qwen3-8B solver (thinking off)
Full-evaluation budget	150	75
Allocator actions	{`FULL_EVAL`, `DISCARD`}	{`FULL_EVAL`, `PARTIAL_EVAL`, `DISCARD`}
Partial-evaluation subset	—	Fixed subset, $k = 8$ (first 8 of 30 problems)
Post-Partial actions	—	{`CONTINUE_TO_FULL`, `STOP`}

Strategy initialization. We initialize S_0 by summarizing 50 example cases, comprising 10 candidate evaluation records from each of five ShinkaEvolve baseline runs with different seeds. The initial selectivity is set to `low`. The cost of obtaining these baseline cases is excluded from the reported search costs.

B.2 ADAS-AIME

ADAS-AIME evolves the Python scaffold around a fixed mathematical-reasoning solver on the 30 AIME 2024 problems. A candidate is a reusable `Agent`: it may query the solver, vary sampling temperature, combine responses, filter or vote over candidate answers, and extract the final integer answer returned to the benchmark harness. The solver weights and benchmark questions are fixed; the evolved object is the reasoning scaffold.

Following the ShinkaEvolve evaluation protocol, each candidate is evaluated over all 30 AIME 2024 problems in three independent runs, and the resulting accuracies are averaged. These three repetitions together count as one Full evaluation in the search budget. For allocation decisions, we use a Partial evaluation on the first 8 of 30 problems using a single evaluation seed. After observing the Partial result, the allocator chooses `CONTINUE_TO_FULL` or `STOP`. The Partial result is used only to decide whether the current proposal deserves the remaining full-evaluation cost; final performance is always measured by the full evaluator. Search and evaluation use the same fixed 30-problem suite. The reported results measure allocation efficiency in optimizing harnesses on this suite.

The example below illustrates a reasoning harness that samples eight solutions at three temperatures, extracts candidate answers, and selects the most frequent answer by majority voting.

Example evolved program: ADAS-AIME

```python
import re
from collections import Counter
from typing import Callable, Optional

class Agent:
    def __init__(
        self,
        query_llm: Callable,
        temperature=0.0,
    ):
        self.query_llm = query_llm
        self.max_calls = 10
    def _extract_answer(self, response: str) -> Optional[int]:
        match = re.search(r'\\boxed\{(\d+)\}', response)
        if match:
            return int(match.group(1))
        lines = response.strip().split('\n')
        for line in reversed(lines):
            line = line.strip()
            if line.isdigit() and 0 <= int(line) <= 999:
                return int(line)
        return None
    def forward(self, problem: str) -> tuple[str, float]:
        system_prompt = (
            "You are an expert mathematician. Solve the following problem step-by-step.
     "
            "Think carefully and verify your steps. "
            "On the final line output only the digits of the answer (0-999). "
            "Provide your final answer enclosed in a LaTeX \\boxed{...} command."
        )
        temperatures = [0.8, 0.8, 0.8, 0.5, 0.5, 0.2, 0.2, 0.2]
        candidates = []
        total_cost = 0.0
        for t in temperatures:
            if len(candidates) >= self.max_calls:
                break
            task_prompt = f"{problem}\n\n"
            response, cost = self.query_llm(
                prompt=task_prompt,
                system=system_prompt,
                temperature=t,
            )
            total_cost += cost
            ans = self._extract_answer(response)
            if ans is not None:
                candidates.append((ans, response))
        if not candidates:
            return "", total_cost
        ans_counts = Counter([c[0] for c in candidates])
        best_ans = ans_counts.most_common(1)[0][0]
        for ans, resp in candidates:
            if ans == best_ans:
```

```
                return resp, total_cost
        return candidates[0][1], total_cost
```

B.3 CIRCLE PACKING

Circle Packing evolves compact Python programs that construct feasible placements of 26 circles in the unit square. The evaluator checks boundary and non-overlap constraints, computes the induced radii, and returns the packing objective. Unlike ADAS-AIME, the evaluator is deterministic and non-LLM; the artifact is a constructive geometric algorithm rather than a language-model scaffold.

We do not use Partial evaluation in this setting. The allocator therefore chooses directly between `FULL_EVAL` and `DISCARD`. This setting isolates adaptive full-evaluation allocation on a deterministic algorithm-design task.

Example evolved program (excerpt): Circle Packing

```
def construct_packing():
    n = 26
    rng = np.random.default_rng(seed=12345)
    initializations = [
        _init_hexagonal(n, rng),
        _init_concentric(n, rng),
        _init_corner_focused(n, rng),
        _init_voronoi(n, rng),
        _init_strategic(n, rng),
    ]
    best_centers = None
    best_radii = None
    best_score = -1.0
    for init_c in initializations:
        best_c, best_r = anneal_centers(init_c, rng, max_iter=35000)
        best_r = radius_expansion(best_c, best_r, max_iters=10)
        best_c, best_r = gradient_refine(best_c, best_r, iterations=14, step_size
     =0.022, momentum=0.78)
        best_r = _lp_max_radii(best_c)
        score = best_r.sum()
        if score > best_score:
            best_score = score
            best_centers = best_c.copy()
            best_radii = best_r.copy()
    best_radii = np.maximum(best_radii, 0.0)
    return best_centers, best_radii
```

C Evolution and Search Analysis

This appendix examines how EvoAlloc's Strategy and Experiences evolve during search, and how these changes reshape the search itself. We first trace their evolution on ADAS-AIME and Circle Packing, then analyze the resulting search trees.

C.1 ADAS-AIME Strategy and Experience Evolution

Strategy trajectory. Figure 5 visualizes the representative ADAS-AIME run analyzed here. The adopted Strategy starts permissive, moves to medium selectivity once sufficient evidence has accumulated, and then passes through two highly selective stages. Its criteria also become more specific over time: the generic guidance of S0 gives way to explicit Partial-evaluation thresholds conditioned on lineage trends. Table 2 lists each stage's Full-evaluation window, selectivity level, and complete Strategy text; the final Experience inventory is reported afterward.

Table 2: **Evolution of the learned allocation Strategy on ADAS-AIME.** Each row lists one Strategy stage with its full-evaluation window, selectivity level, and complete Strategy text, with "smoke" replaced by "partial evaluation" for consistency with the main text.

Stage	Full window	Selectivity	Allocation strategy
S0	1–19	Low	Allocate evaluator calls using the available evidence while preserving useful uncertainty. Preserve evaluator calls without hiding promising or informative candidates.
S1	20–31	Medium	Prefer direct full evaluation for structural or prompting redesigns with at least 3 of 8 partial-evaluation successes. Use partial evaluation for parameter-level changes or unclear lineages, and reserve discarding for poor partial-evaluation execution or lineages already at the current best without a novel mechanism.
S2	32–44	High	Reserve full evaluation for strong structural redesigns with at least 4 of 8 partial-evaluation successes, especially when the lineage is improving or the parent is near the current best. Continue after partial evaluation only under the same evidence; otherwise stop early, and discard proposals with few partial-evaluation successes from flat or regressive lineages more aggressively.
S3	45–75	High	Reserve full evaluation for structural redesigns with at least 4 of 8 partial-evaluation successes and clear lineage improvement. Use partial evaluation for parameter-level or unclear-lineage proposals, but continue only with both strong partial-evaluation evidence and an improving lineage. Aggressively discard proposals with at most 2 of 8 partial-evaluation successes when the lineage is flat or declining or the change lacks structural novelty.

Final learned Experiences. The Experience inventory records recurring search situations and the outcome tendencies distilled from them during the run. Each entry specifies an applicability condition, an expected tendency, and its supporting and contradicting evidence. Unlike the run-level Strategy, Experiences retain local, case-grounded information that can be reused when similar search states reappear. Conditions that refer to Partial-evaluation results apply only after Partial evaluation has been observed. As in Table 2, "smoke" in the original text is replaced by "partial evaluation".

Experience 1 *active*

APPLICABILITY
Proposals with at most 2 of 8 partial-evaluation successes, especially when the lineage is flat or declining, even when the proposed structural change appears promising.

EXPECTED TENDENCY
They tend to yield low or non-improving full scores regardless of design ambition.

EVIDENCE
15 support / 3 counter.

Experience 2 *active*

APPLICABILITY
Targeted deterministic solvers with at least 4 of 8 partial-evaluation successes when the lineage is improving and the parent is at or near the current best.

EXPECTED TENDENCY
They can improve over the parent and sometimes establish a new best when the fix targets a known failure mode.

EVIDENCE
2 support / 0 counter.

Experience 3 *stale*

APPLICABILITY
Targeted deterministic solvers for recurring AIME patterns with at most 2 of 8 partial-evaluation successes, especially on a flat or declining lineage.

EXPECTED TENDENCY
They may still improve over the parent when the structural fix targets a known blind spot.

EVIDENCE
2 support / 2 counter.

Experience 4 *stale*

APPLICABILITY
Targeted deterministic solvers with at least 4 of 8 partial-evaluation successes, especially on a flat or declining lineage.

EXPECTED TENDENCY
They often plateau at or below the current best despite strong partial-evaluation results and structural novelty.

EVIDENCE
4 support / 1 counter.

C.2 Circle Packing Strategy and Experience Evolution

Strategy trajectory. Figure 8 shows a representative Circle Packing run. Its Strategy follows a progression in selectivity similar to ADAS-AIME, but without Partial evaluation, so each decision is made directly between Full evaluation and discard. The allocator first evaluates proposals broadly, then becomes increasingly selective as evidence accumulates and the search matures. Table 3 lists the corresponding Strategy stages.

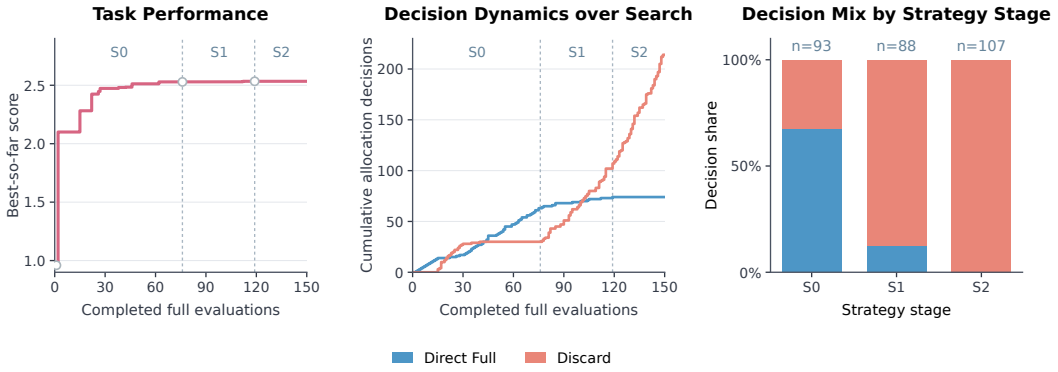


Figure 8: **Evolution of allocation behavior on Circle Packing.** A representative 150-full-evaluation run (seed 3) showing best-so-far performance and allocation decisions across S0–S2. Allocation panels show decisions before counterfactual exploration, with explored proposals counted by their original allocation decision. Dashed lines mark the adoption of validated Strategies.

Table 3: **Evolution of the learned allocation Strategy on Circle Packing.** Each row lists one Strategy stage with its full-evaluation window, selectivity level, and complete Strategy text.

Stage	Full window	Selectivity	Allocation strategy
S0	1–76	Low	Begin with low selectivity. Early proposal outcomes are highly variable, and surface plausibility does not reliably distinguish failures from improvements. Evaluate broadly until grounded outcomes establish a repeatable basis for discarding, then tighten allocation gradually. Use the limited evaluation budget efficiently over the full search.
S1	77–119	Medium	Begin to favor targeted evaluation. Retain broad coverage for proposals with strong lineage improvement trends or structural novelty, but increasingly discard repeated marginal variants or invalid proposals without structural divergence. Prioritize full evaluation for candidates that extend a promising new-best lineage or introduce a substantially new initialization or optimization strategy.
S2	120–150	High	Strongly favor discarding repeated marginal variants on plateaued lineages or motifs associated with invalid and non-improving cases. Reserve full evaluation primarily for structurally novel strategies with promising lineage evidence, or for candidates directly extending a new-best lineage and plausibly escaping a local optimum.

Final learned Experiences. The final inventory summarizes recurring proposal patterns and their observed outcomes. Descriptions such as "valid but non-improving" characterize past evaluated cases, not conditions to be checked on a new candidate. These Experiences inform Strategy revision and expectations about related proposals through visible code changes and known lineage outcomes. Allocation uses only information available at decision time, without access to the current candidate's Full outcome (Appendix E.2).

Experience 1 *active*

APPLICABILITY
Valid but substantially non-improving candidates, especially minor perturbations of invalid or low-quality structures.

EXPECTED TENDENCY
Such candidates rarely improve without structural innovation.

EVIDENCE
9 support / 0 counter.

Experience 2 *active*

APPLICABILITY
Invalid outputs associated with structural flaws, such as overlap or boundary violations, often following aggressive or poorly constrained edits.

EXPECTED TENDENCY
Invalid outcomes rarely precede improvement without corrective refinement.

EVIDENCE
28 support / 0 counter.

Experience 3 *active*

APPLICABILITY
Candidates that improve on a valid parent, even when their absolute quality remains low.

EXPECTED TENDENCY
Local parent improvement can reveal useful structural motifs and may precede larger gains after further refinement.

EVIDENCE
58 support / 9 counter.

Experience 4 *stale*

APPLICABILITY
Rare new-best outcomes that emerge after repeated invalid or low-scoring attempts.

EXPECTED TENDENCY
Such outcomes are not reliably predictable from prior local scores and may require full evaluation to confirm.

EVIDENCE
9 support / 1 counter.

C.3 Search-Tree Analysis

Figure 9 complements Section 5.4 with Circle Packing search trees. With the same seed and 150 Full evaluations, EvoAlloc considers 302 proposals, compared with 150 for ShinkaEvolve. Since Circle Packing has no Partial evaluation, this expansion comes entirely from deciding which proposals receive Full evaluation. Highlighted edges trace the lineages of the best programs.

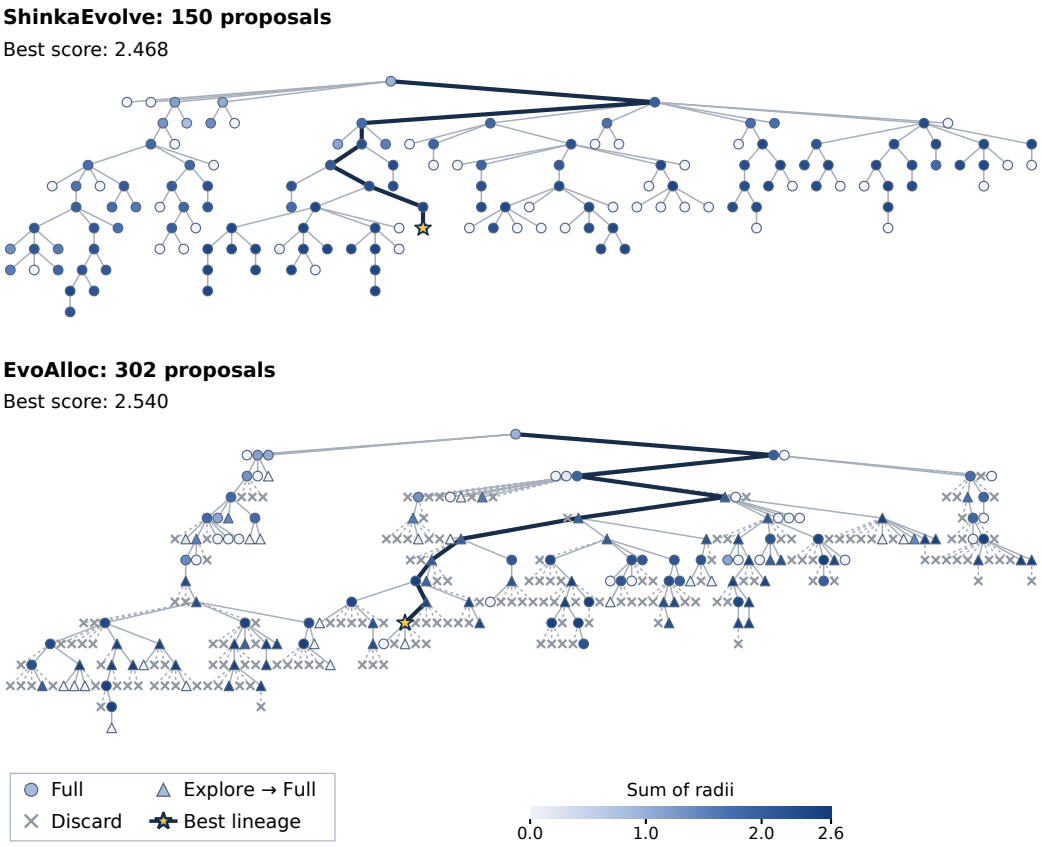


Figure 9: **Circle Packing search trees.** Both methods complete 150 full evaluations with the same seed (seed 1). Shapes encode allocation decisions; colors indicate the observed sum of radii. Gray crosses denote proposals without Full outcomes; stars mark the best programs, and highlighted edges trace their lineages.

D Experimental Details and Additional Results

This appendix describes the baseline and ablation configurations and reports detailed performance, resource costs, and allocation breakdowns. Implementation settings and benchmark protocols are given in Appendix B.

D.1 Baselines and Comparison Configuration

All methods run within the same ShinkaEvolve framework, using its default configuration and benchmark evaluation protocols unless stated otherwise in Appendix B. They differ only in how evaluation resources are allocated after a candidate is generated. Following the framework's default behavior, a candidate that does not ultimately receive Full evaluation is discarded and cannot be selected as a parent. Allocation therefore determines which candidates enter the archive, and different methods may consequently induce different proposal streams over the course of a run; we treat this interaction as part of the closed-loop allocation problem. While the default configuration uses a single seed, we run each method with five seeds and report averages for more reliable comparisons.

ShinkaEvolve. ShinkaEvolve (Lange et al., 2026) is the program-evolution framework underlying all methods and serves as the full-evaluation reference, in which every generated candidate receives Full evaluation. We retain its main search mechanisms, including an ensemble of proposer LLMs sampled at multiple temperatures, a meta model that periodically reflects on evaluated programs to guide subsequent proposals, and novelty filtering that combines embedding similarity with an LLM novelty judge to reject near-duplicate candidates. The models used for each role are listed in Table 1. We rerun ShinkaEvolve with the same locally served open-weight search models used by EvoAlloc and the other baselines (Table 1). This differs from the proprietary model ensemble used in the original study, so the reported scores should be interpreted within our shared model configuration rather than as a reproduction of its published results. Our comparisons assess evaluation allocation under a common search backbone and matched Full-evaluation budgets.

One-Step LLM Allocator. This baseline isolates the effect of EvoAlloc's persistent allocation state while matching its allocator, action space, counterfactual exploration, and decision context. Each decision is conditioned on the same search information available to EvoAlloc, including recent cases, retrieved similar cases, and the current search state. The baseline differs only in maintaining neither an explicit Strategy nor distilled Experiences; instead, it maps the available context directly to an allocation action at each step. This comparison evaluates the benefit of explicitly consolidating search history into reusable Experiences and an evolving Strategy, beyond directly conditioning an LLM allocator on the search history.

Lineage-UCB. Lineage-UCB represents classical adaptive allocation based on a fixed statistical rule, in the spirit of bandit-based methods. It uses a binary action space of Full evaluation or discard on both benchmarks. For each candidate, it estimates the improvement tendency of its local lineage using a Beta posterior initialized as $\mathrm{Beta}(1, 1)$ and updated from previously observed outcomes, with weight 1 for direct relatives and 0.5 for second-level relatives. A candidate receives Full evaluation if the posterior mean plus one standard deviation is at least 0.5, and is otherwise discarded. The allocator requires no LLM calls. This comparison isolates the benefit of learned, structured allocation guidance beyond online adaptation under a fixed statistical rule.

RPM. We include RPM (Foster et al., 2026) as a recent LLM-based proposal-selection baseline that addresses the closely related problem of deciding which candidates merit computationally expensive evaluation. We adapt its inference-time selector to our setting. At each selection step, 15 candidates are generated from the same parent and compared through a seeded, randomly ordered knockout tournament, with the winner receiving Full evaluation and all others discarded. The batch size of 15 follows the end-to-end configuration in the original RPM study. Each pairwise comparison is conditioned on the two candidates' plans and code, together with up to 10 previously evaluated programs retrieved from the search history. We use Qwen3-Coder-Next as the judge with temperature 0 and a maximum of 4,096 output tokens. Unlike EvoAlloc, RPM performs batch-level selection among competing candidates and follows a fixed selection rule throughout search. LLM tokens

consumed by all generated candidates, including those not selected for evaluation, are included in the total cost.

Ablations. Section 5.3 compares EvoAlloc with variants using one-step allocation, fixed allocation guidance, and no counterfactual exploration. **One-Step Alloc** corresponds to the One-Step LLM Allocator described above, removing both the persistent Strategy and distilled Experiences. **w/o Evolution** uses fixed custom allocation guidance: a medium-selectivity Strategy and seeded Experiences distilled from baseline run examples, with the Strategy manually refined into reasonable allocation criteria. This guidance remains fixed throughout search. **w/o Exploration** sets $\rho = 0$, preventing discarded or stopped candidates from receiving exploratory Full evaluation and thus leaving their Full outcomes unobserved.

D.2 Detailed Performance

Table 4 reports final performance and normalized trajectory AUC as an auxiliary anytime measure; matched-target cost remains the primary efficiency metric. For each seed, we normalize the area under the best-so-far curve by the range length, using the full evaluation budget for eval-budget AUC and the common token range covered by all methods and seeds for token-budget AUC (0.62–55.1M tokens on ADAS-AIME and 0.27–4.68M on Circle Packing). We report the mean and standard deviation across five seeds, with standard deviations computed under the population convention to match the shaded bands in Figure 3.

Table 4: **Final performance and auxiliary trajectory AUC.** Five-seed mean ± one standard deviation; bold denotes the best method per benchmark and column. AUC is the normalized area under the best-so-far curve over the evaluation or token budget; higher is better.

Benchmark	Method	Performance	Eval-budget AUC	Token-budget AUC
ADAS-AIME	ShinkaEvolve	35.33 ±2.57	31.29 ±1.95	30.24 ±2.82
	One-Step Alloc	34.89 ±1.94	31.40 ±1.34	32.45 ±1.49
	Lineage-UCB	34.00 ±0.89	30.88 ±0.53	32.32 ±1.07
	RPM	36.22 ±1.13	32.49 ±0.69	30.31 ±0.94
	EvoAlloc (ours)	**39.56 ±6.80**	**33.69 ±3.33**	**33.15 ±2.89**
Circle Packing	ShinkaEvolve	2.343 ±0.254	2.172 ±0.239	2.245 ±0.257
	One-Step Alloc	2.429 ±0.219	2.261 ±0.222	2.251 ±0.266
	Lineage-UCB	2.408 ±0.131	2.181 ±0.156	2.213 ±0.147
	RPM	2.390 ±0.186	2.246 ±0.151	1.553 ±0.277
	EvoAlloc (ours)	**2.547 ±0.030**	**2.389 ±0.043**	**2.436 ±0.024**

D.3 Resource and Token Costs

Table 5 summarizes the Full evaluations and total LLM tokens needed to reach matched performance targets. For each method, we align the five seed trajectories by completed Full evaluations and average both their best-so-far scores and cumulative token usage at each evaluation index. We then identify the first index at which the mean score reaches the target. The reported costs are this Full-evaluation index and the corresponding mean cumulative token usage.

Target matching is therefore performed after averaging across seeds, rather than averaging each seed's individual first-hit cost. In particular, the reported token cost is not the first-hit token budget of the token-aligned mean performance curve shown in Figure 3. It is the mean cumulative token usage at the target-reaching Full-evaluation index, as used consistently in Table 5 and the bottom panels of Figure 3.

The reference final target is ShinkaEvolve's five-seed mean final score. Let Q_{initial} and Q_{final} denote the first and final values of its mean best-so-far curve. The 95%-gain target is

$$q_{95} = Q_{\text{initial}} + 0.95\,(Q_{\text{final}} - Q_{\text{initial}}), \tag{3}$$

corresponding to 95% of ShinkaEvolve's improvement over its initial score.

Budget accounting. All Full evaluations performed during online search count toward the same search budget, including those from warm-up, counterfactual exploration, and Strategy validation; shared evaluations are counted once. Recorded LLM usage includes proposal generation, benchmark evaluation, and allocator calls for resource allocation, self-evolution, and shadow validation, and Partial evaluations are counted whether a proposal stops or continues. For RPM, allocator costs are the tokens spent on pairwise comparisons; Lineage-UCB requires no LLM calls for allocation. Table 6 reports costs at the full evaluation budget. At this budget, EvoAlloc consumes more total tokens than ShinkaEvolve, since the evaluations it withholds are reinvested in examining additional proposals (Section 5.4). Its allocator accounts for 1.02M tokens on ADAS-AIME and 1.67M on Circle Packing. Total usage need not scale linearly with the number of proposals, as generated programs differ in their evaluation costs. The savings in Table 5 therefore reflect how quickly each method reaches a given performance level, rather than its total consumption once the budget is exhausted.

Table 5: **Resources to matched performance targets.** First-hit costs read from the five-seed mean best-so-far curves, with mean cumulative token usage taken at the corresponding Full-evaluation index. Parentheses show changes relative to ShinkaEvolve; dashes indicate that the mean curve does not reach the target within the observed evaluation budget.

	Reference final target (100%)		**95%-gain target**	
Method	**Full evals ↓**	**LLM tokens (M) ↓**	**Full evals ↓**	**LLM tokens (M) ↓**
ADAS-AIME	target: ≥ 35.33		target: ≥ 34.49	
ShinkaEvolve	69	89.8	51	63.6
One-Step Alloc	–	–	66 (+29%)	77.7 (+22%)
Lineage-UCB	–	–	–	–
RPM	66 (−4%)	140.1 (+56%)	57 (+12%)	117.2 (+84%)
EvoAlloc (ours)	**28 (−59%)**	**35.3 (−61%)**	**26 (−49%)**	**32.7 (−49%)**
Circle Packing	target: ≥ 2.343		target: ≥ 2.274	
ShinkaEvolve	148	6.1	70	2.0
One-Step Alloc	84 (−43%)	5.0 (−18%)	44 (−37%)	1.9 (−4%)
Lineage-UCB	95 (−36%)	3.5 (−42%)	76 (+9%)	2.7 (+36%)
RPM	67 (−55%)	41.4 (+582%)	40 (−43%)	19.0 (+860%)
EvoAlloc (ours)	**26 (−82%)**	**0.7 (−89%)**	**24 (−66%)**	**0.6 (−70%)**

D.4 Allocation Breakdown

Table 6 shows how each method spends the same Full-evaluation budget. At this budget, EvoAlloc considers 136.8 proposals per seed on ADAS-AIME and 243.0 on Circle Packing, compared with 75.0 and 150.0 for ShinkaEvolve. Proposal counts include all candidates that reach the allocation stage and complete their allocation episode, with initialization and warm-up candidates counted as Direct Full. Allocation percentages are computed per seed over the same candidates and record each candidate's original decision rather than its evaluation outcome: candidates evaluated for Strategy validation remain in their original category, and Explore→Full counts only candidates revealed by counterfactual exploration. For RPM, we report the nominal counts prescribed by its protocol, with 15 proposals per Full evaluation of which one is selected, while its token costs are measured from the actual runs.

Table 6: **Token costs and allocation decisions.** Five-seed averages at the full evaluation budget; RPM allocation statistics follow its nominal batch protocol (Section D.4). Allocator tokens include resource allocation, self-evolution, and shadow validation; for RPM, they are the tokens spent on pairwise selection. Dashes denote methods without an LLM-based allocator or allocation paths a method does not use. Allocation categories are mutually exclusive and sum to 100% within each method, up to rounding, with percentages computed per seed and then averaged. Percentages describe allocator decisions rather than all executed evaluations; candidates receiving additional Full evaluations during validation are counted under their original decision categories.

	Token & proposal cost			Allocation breakdown (%)				
Method	**Total tokens (M)**	**Allocator/selector (M; % of total)**	**Proposal count**	**Direct Full**	**Partial →Full**	**Partial →Stop**	**Direct Discard**	**Explore →Full**
ADAS-AIME								
ShinkaEvolve	98.44	–	75.0	100.0	–	–	–	–
One-Step Alloc	92.53	0.03 ($< 0.1\%$)	81.2	45.7	43.4	7.5	0.0	3.4
Lineage-UCB	84.93	–	95.4	78.7	–	–	21.3	–
RPM	178.27	10.98 (6.2%)	1125.0	6.7	–	–	93.3	–
EvoAlloc (ours)	128.76	1.02 (0.8%)	136.8	6.2	26.7	19.1	29.2	18.8
Circle Packing								
ShinkaEvolve	6.11	–	150.0	100.0	–	–	–	–
One-Step Alloc	10.57	0.38 (3.6%)	266.8	41.3	–	–	41.6	17.1
Lineage-UCB	6.61	–	190.4	78.8	–	–	21.2	–
RPM	121.31	29.64 (24.6%)	2250.0	6.7	–	–	93.3	–
EvoAlloc (ours)	10.85	1.67 (16.1%)	243.0	25.5	–	–	58.5	16.0

D.5 Additional Ablation Experiments

Ablation of Random Allocation. To examine whether content-independent filtering can reproduce EvoAlloc's gains, we evaluate Random Allocation on Circle Packing. After warm-up, this allocator discards each candidate with probability 0.5, independently of its content, and otherwise requests Full evaluation. We compare five-seed results under a budget of 150 Full evaluations per run, using the main-experiment EvoAlloc results as the reference.

EvoAlloc achieves a final sum of radii of 2.547 ± 0.030, compared with 2.404 ± 0.184 for Random Allocation. Figure 10 also shows higher mean best-so-far performance for EvoAlloc against both Full evaluations and cumulative LLM tokens. Randomly withholding evaluations at this fixed probability therefore does not reproduce EvoAlloc's observed performance.

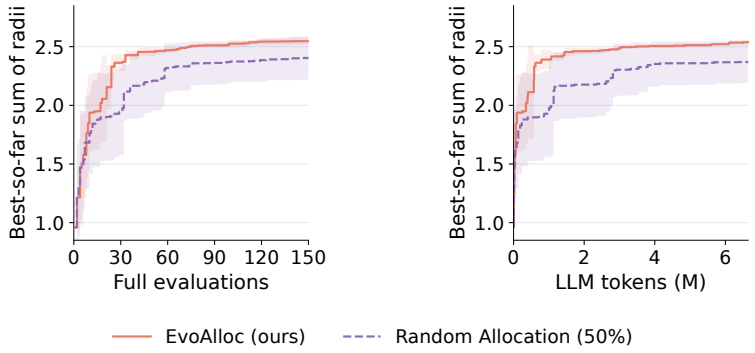


Figure 10: **Ablation with random allocation on Circle Packing.** Random Allocation discards candidates with probability 0.5 after warm-up. Best-so-far performance is shown against Full evaluations (left) and cumulative LLM tokens (right). Lines show five-seed means, and shading denotes ± 1 standard deviation. Token curves use the common observed range across methods and seeds.

Ablation of Exploration Feedback. The **w/o Exploration** ablation in Section 5.3 removes exploratory evaluations together with the feedback they provide. To examine the feedback contribution while retaining exploration, we introduce **w/o Exploration Feedback**. This variant keeps the exploration probability at $\rho = 0.3$: explored candidates still receive Full evaluation, enter the search archive, and can improve the best-so-far score or serve as parents. However, their outcome cases are withheld from subsequent allocation decisions, Experience updates, and Strategy revisions. The ablation thus removes explicit case feedback while preserving explored candidates' participation in program search.

We compare five-seed results with the main-experiment EvoAlloc reference, using budgets of 150 Full evaluations on Circle Packing and 75 on ADAS-AIME. On Circle Packing, w/o Exploration Feedback achieves 2.446 ± 0.110, compared with 2.547 ± 0.030 for EvoAlloc; the corresponding trajectories are shown in Figure 11. On ADAS-AIME, the final accuracies are $(32.22 \pm 4.10)\%$ and $(39.56 \pm 6.80)\%$, respectively. The lower mean performance without explicit feedback is consistent with exploration helping not only by recovering candidates, but also by informing subsequent allocation.

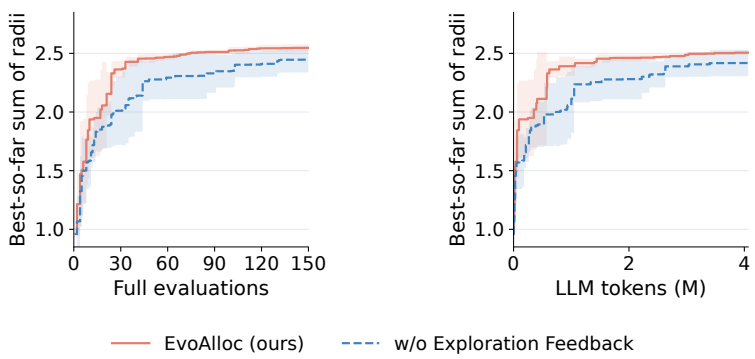


Figure 11: **Ablation of exploration feedback on Circle Packing.** The ablated variant retains exploratory evaluations and archive updates but withholds their explicit outcome-case feedback from allocation and self-evolution. Best-so-far performance is shown against Full evaluations (left) and cumulative LLM tokens (right). Lines show five-seed means, and shading denotes ± 1 standard deviation. Token curves use the common observed range across methods and seeds.

D.6 Exploration Outcomes across Strategy Stages

We analyze candidates with Full outcomes obtained through random counterfactual exploration ($\rho = 0.3$) from five runs per benchmark. We count only exploration following DISCARD or STOP, excluding warm-up and shared-validation evaluations. Candidates are grouped by the Strategy active at decision time: S_0 is the initial Strategy, and subsequent indices denote successive adopted revisions within each run. A candidate is *new-best* if its Full score exceeds the decision-time best-so-far score, and *improved-parent* if it exceeds its parent's score; the latter includes new-best candidates.

Table 7 shows that these recoveries are not confined to the initial Strategy: all six new-best candidates on ADAS-AIME occur under S_2 or S_3, while Circle Packing recovers 15 under S_2. Exploration thus reveals missed improvements even after multiple Strategy revisions, exposing outcomes that the allocator's decisions alone would leave unobserved. Stage coverage differs across runs, with only one run per benchmark reaching S_3, so the pooled rates should not be interpreted as a common temporal trend.

Table 7: **Outcomes recovered by counterfactual exploration across Strategy stages.** Counts are pooled across five seeds within each benchmark. Percentages use the number of explored candidates in each row as the denominator, rather than averaging per-seed rates. Improved-parent includes new-best.

Benchmark	Strategy stage	Runs reaching the stage	Explored candidates	New-best	Improved-parent
ADAS-AIME	S_0	5	10	0 (0.0%)	3 (30.0%)
	S_1	5	35	0 (0.0%)	4 (11.4%)
	S_2	4	72	5 (6.9%)	18 (25.0%)
	S_3	1	15	1 (6.7%)	8 (53.3%)
	Total	-	132	6 (4.5%)	33 (25.0%)
Circle Packing	S_0	5	13	2 (15.4%)	7 (53.8%)
	S_1	5	80	5 (6.3%)	22 (27.5%)
	S_2	5	200	15 (7.5%)	62 (31.0%)
	S_3	1	3	0 (0.0%)	0 (0.0%)
	Total	-	296	22 (7.4%)	91 (30.7%)

D.7 Strategy Validation Details

Shared evaluation. During validation, the incumbent Strategy S and the challenger $\widetilde{S}$ are applied to the same online proposals with the same archive, Experiences, and decision-time information. EvoAlloc executes the union of their requested evaluations once, charges them to the search budget, and shares the resulting evidence with both Strategies. A candidate requested for Full evaluation only by the challenger is therefore still evaluated, so validation is not restricted to outcomes selected by the incumbent. Outcomes of candidates that neither Strategy sends to Full evaluation are revealed only through counterfactual exploration (Section 4.3), which also supplies evidence for skipped candidates outside validation windows.

Consequence-based comparison. Let I denote the proposals in a validation window on which the two Strategies select different allocation paths and the Full outcome is known, whether requested by either Strategy or revealed by exploration; a window closes once $|I| = 2$. For $T \in \{S, \widetilde{S}\}$, let $m_{\text{best}}(T)$ count proposals in I whose Full score exceeds the best score in the shared archive at decision time but whose allocation path under T would leave that outcome hidden. Let $n_F(T)$ and $n_P(T)$ count the Full and Partial evaluations requested under T on these proposals, rather than those executed for the union. The challenger is adopted if

$$Z(T) = \big(m_{\text{best}}(T),\, n_F(T),\, n_P(T)\big)$$

is lexicographically smaller for $\widetilde{S}$ than for S; a complete tie retains the incumbent, and the last component is omitted when no Partial evaluator is available. This establishes a preference over realized allocation consequences within a finite window, not a guarantee of improvement in $\mathbb{E}[Q^{\star}_{\mathcal{A}}(B)]$; continued search supplies evidence for subsequent Strategy revisions.

Observed validation outcomes. Table 8 summarizes how often proposed challengers were adopted across both benchmarks, together with the distribution of adoption criteria. Each validation window compares a challenger with the incumbent on shared proposals; completed windows are those that reached a decision to adopt the challenger or retain the incumbent. The reported adoptions were driven by fewer Full or Partial evaluations, while matching the incumbent's missed-new-best count within the corresponding validation windows. Validation thus primarily favored more selective use of evaluation resources, refining how much evaluation to allocate to each candidate.

Table 8: Strategy validation outcomes aggregated over five seeds per benchmark. The final three columns count adoptions decided by fewer missed new-best candidates (m_{best}), fewer Full evaluations (n_F), or fewer Partial evaluations (n_P).

				Adoption criterion		
Benchmark	Proposed challengers	Completed windows	Adopted challengers	m_{best}	n_F	n_P
ADAS-AIME	23	20	10	0	7	3
Circle Packing	44	42	11	0	11	0

E PROMPTS AND CONTEXT

This appendix details the inputs, prompts, and representative outputs of the two LLM calls in EvoAlloc: resource allocation for each candidate (Section 4.1), and self-evolution, which updates Experiences and proposes Strategy revisions (Section 4.2).

E.1 INPUT CONTEXT

Each LLM call in EvoAlloc is conditioned on a structured context assembled from the ongoing search, summarized in Table 9. Resource allocation and self-evolution both receive the benchmark setup, the current allocation guidance, recent cases, and a summary of search progress. Resource allocation additionally receives the current candidate, its search context, and historical cases retrieved by similarity; together with search progress, these form the decision-time context c_t of Section 3.1. Since allocation decisions are made before the candidate is fully evaluated, its own Full outcome is never part of this context.

Table 9: **Input context for resource allocation and self-evolution.**

Context	Information
Shared by resource allocation and self-evolution	
Benchmark	Evaluation protocol and available actions
Allocation guidance	Current Strategy and Experiences
Recent cases	Recently observed cases, as case cards (Appendix E.4)
Search progress	Budget usage and recent decision counts
Additional context for resource allocation	
Candidate	Candidate description and code changes relative to its parent
Search context	Lineage scores, the parent's gap to the current best, and sibling outcomes
Retrieved cases	Similar historical cases, with their observed outcomes
Partial evidence	Partial-evaluation result, for the continuation decision only

E.2 RESOURCE ALLOCATION PROMPT

The prompt below guides resource allocation using the current Strategy, relevant Experiences, and available search context. Representative structured outputs illustrate the initial allocation and post-Partial continuation decisions.

Resource allocation prompt

```
You are the resource allocator of EvoAlloc, a resource-allocation system in a
     self-improving program search loop. EvoAlloc aims to improve sample efficiency over
     the full run by allocating evaluations where they have the greatest search value.

Through self-evolution, EvoAlloc learns reusable search experiences from observed
     outcomes and maintains an allocation strategy for future resource allocation. Use
     this strategy as the primary guidance, together with relevant experiences and search
     context, to choose among:

- FULL_EVAL: run the full benchmark evaluation and observe its result.
- PARTIAL_EVAL: obtain limited evidence, then choose CONTINUE_TO_FULL or STOP.
- DISCARD: skip full evaluation and leave its result hidden.

Judge the proposal's marginal search value against preserving the same resources for
     future search. Consider both its current evidence and whether evaluating it would
     preserve a branch with meaningful downstream search value.

Use the allocation strategy as the run-level default, and adjust the decision when
     current grounded experience is materially relevant.

Apply only guidance that is usable at decision time; ignore any condition that requires
     the current proposal's unknown Full outcome.

- Use PARTIAL_EVAL only when at least two plausible partial-evaluation observations would
     support different final actions. If likely observations would all support the same
     final action, take it now. Partial evaluation is an information-acquisition action,
     not a safe compromise or an automatic step before Full evaluation.
```

```
- Interpret partial evaluation according to what it reveals: successful execution alone
    does not justify CONTINUE_TO_FULL, and low partial performance alone does not
    justify STOP. A reproducible candidate-side execution failure is strong negative
    evidence because Full evaluation runs the same candidate unchanged.
```

Post-Partial continuation. When Partial evaluation is requested, its verified observation is appended to the original allocation context and the allocator makes a second decision using the same proposal, Strategy, Experiences, and search context.

Post-Partial continuation prompt

```
The partial evaluator returned the following verified observation:
{partial_observation}

Complete the same allocation episode using the original proposal, allocation strategy,
    experiences, and search context.

Return only one valid JSON object:
{continuation_schema}
```

The outputs use structured JSON. Ellipses in the abbreviated examples denote generated text.

Resource allocation: initial decision

```
{
  "action": "PARTIAL_EVAL",
  "matched_experience_ids": ["e1", "e3"],
  "reason": "...",
  "partial_request": {"purpose": "..."}
}
```

Resource allocation: post-Partial

```
{
  "action": "CONTINUE_TO_FULL",
  "reason": "...",
  "used_experience_ids": ["e1"]
}
```

E.3 Self-Evolution Prompts

The prompts below specify separate updates to Experiences and the Strategy. Experience updates maintain case-grounded evidence, while Strategy updates propose challengers for shadow validation.

Experience operations. `ADD` introduces a new Experience with initial status `tentative`. `UPDATE` attaches supporting or contradicting evidence to an existing Experience and may change its status, but cannot rewrite its content, so accumulated evidence always refers to a fixed statement. `DELETE` removes an Experience. Rewriting an Experience therefore requires deleting it and adding a new one, and no operation is performed when evidence is insufficient. Each Experience has one of three statuses: `tentative` (limited or mixed evidence), `active` (consistent evidence), or `stale` (retained as a historical record but no longer guiding current decisions). Unlike `DELETE`, marking an Experience `stale` keeps it in memory.

Experience update prompt

```
Update only the evolving search experiences.

Experiences are descriptive. Each contains one recurring pattern and one qualitative
    outcome tendency, separated into applicability visible at decision time and observed
    tendency. It must remain identifiable and useful across future proposals, while
    retaining enough mechanism-level detail to distinguish the pattern. Experiences do
    not prescribe allocation actions.

Full outcomes, including those revealed by exploration, are grounded evidence. Explored
    outcomes provide grounded evidence for allocation decisions whose Full outcomes
    would otherwise remain hidden. Partial evaluation provides limited evidence;
    unexplored DISCARD and STOP cases have unknown Full outcomes.
```

Strategy update prompt

```
Update only the allocation strategy.

The strategy is prescriptive. It is the allocator's run-level posture for balancing the
     available allocation actions as evidence and budget change, rather than encoding a
     decision tree.

Reconsider the strategy using recent outcomes, including those revealed by exploration,
     together with accumulated experiences and broader run context, then ALWAYS propose
     the allocation strategy you would use right now as a challenger.
```

Self-evolution: Experience update

```
{
  "experience_operations": [{
      "operation": "UPDATE",
      "experience_id": "e1",
      "status": "active",
      "support_case_ids":
        ["Case-5", "Case-10"],
      "counter_case_ids": [],
      "rationale": "..."
  }]
}
```

Self-evolution: Strategy update

```
{
  "allocation_strategy_update": {
    "operation": "REVISE",
    "selectivity": "medium",
    "allocation_strategy": "...",
    "rationale": "...",
    "evidence_experience_ids": ["e1"]
  }
}
```

E.4 Case Card Examples

Each observed candidate is summarized as a case card, which records the proposal, its search context, the allocation decision, and the observed outcomes. Case cards provide the recent and retrieved cases in the input context of resource allocation and self-evolution (Table 9). The examples below show two ADAS-AIME case cards: Partial evaluation followed by Full evaluation, and Partial evaluation followed by `STOP` and later counterfactual exploration.

ADAS-AIME: Partial evaluation followed by Full evaluation

```
Case-11

Proposal:
- Name: self_consistency_ensemble_verification

Search context:
- Current best: 32.222
- Lineage scores: 21.111
- Parent position: below current best
- Lineage trend: insufficient observed history
- Earlier siblings: 1 observed; scores 32.222
- Sibling outcomes: 1/1 improved over parent; 1/1 reached current best
- Decision-time experience matches: e1

Observed allocation path and outcomes:
- Allocation path: PARTIAL_EVAL -> CONTINUE_TO_FULL
- Partial: 3/8 passed; execution completed.
- Full eval: 40.000, valid new-best
```

ADAS-AIME: Partial–STOP followed by counterfactual exploration

```
Case-67

Proposal:
- Name: rigorous_verify_synthesize

Search context:
- Current best: 33.333
- Lineage scores: 25.556 -> 28.889 -> 27.778
- Parent position: below current best
- Lineage trend: mixed
- Earlier siblings: 1 observed; scores 27.778
- Sibling outcomes: 0/1 improved over parent; 0/1 reached current best
- Decision-time experience matches: e2

Observed allocation path and outcomes:
- Allocation path: PARTIAL_EVAL -> STOP -> EXPLORE_EVAL
- Partial: 3/8 passed; execution completed.
- Full eval: 28.889, valid improved parent
```