Escher-Loop: Mutual Evolution by Closed-Loop Self-Referential Optimization

Ziyang Liu [1 2 †] **Xinyan Guo** [1 3 †] **Xuchen Wei** [1 †] **Han Hao** [4 *] **Liu Yang** [5 *]

Abstract

While recent autonomous agents demonstrate impressive capabilities, they predominantly rely on manually scripted workflows and handcrafted heuristics, inherently limiting their potential for open-ended improvement. To address this, we propose *Escher-Loop*, a fully closed-loop framework that operationalizes the mutual evolution of two distinct populations: *Task Agents* that solve concrete problems, and *Optimizer Agents* that recursively refine both the task agents and themselves. To sustain this self-referential evolution, we propose a dynamic benchmarking mechanism that seamlessly reuses the empirical scores of newly generated task agents as relative win-loss signals to update optimizers' scores. This mechanism leverages the evolution of task agents as an inherent signal to drive the evaluation and refinement of optimizers without additional overhead. Empirical evaluations on mathematical optimization problems demonstrate that Escher-Loop effectively pushes past the performance ceilings of static baselines, achieving the highest absolute peak performance across all evaluated tasks under matched compute. Remarkably, we observe that the optimizer agents dynamically adapt their strategies to match the shifting demands of high-performing task agents, which explains the system's continuous improvement and superior late-stage performance.

1. Introduction

Despite the rapid development of autonomous agents (Yang et al., 2023; Anthropic, 2025; Luo et al., 2025), the majority of current systems remain limited to human-designed tools and manually scripted workflows (Yao et al., 2023; Shinn et al., 2023). These systems are often engineered for specific tasks, relying heavily on human priors to navigate complex problem spaces (Jimenez et al., 2024; Wang et al., 2024; Yang et al., 2024b). While effective in narrow domains, this reliance on handcrafted heuristics inherently limits their generalization and caps their potential for autonomous growth. This trend stands in direct opposition to the "bitter lesson" (Sutton, 2019): *history has shown that methods leveraging scalable computation and adaptive search invariably outperform those built on human-engineered intuition.* While the pursuit of recursive self-improvement to overcome these limits has a rich history (Good, 1966; Schmidhuber, 2003; Zhang et al., 2026a; Wang et al., 2026; Zhang et al., 2026b), we must confront a fundamental question: *what is the most foundational and indispensable capability that enables true intelligence?*

We argue that the core of intelligence lies not in the static mastery of a task, but in the *dynamic ability to optimize both the task solution and the optimizer itself.* Consider how humans undertake a sophisticated project: we do not act as a static agent that merely provides solutions. Instead, we engage in a dual-layered refinement, constantly improving the *task solution* we are building while simultaneously sharpening our capability in *optimization*. An engineer does not just write code; they refine their mental models and debugging strategies as the project grows in complexity (Flavell, 1979; Schmidhuber, 1987). Similarly, a successful organization does not just focus on its immediate output; it evolves its internal structure and decision-making logic to function more effectively (Argyris, 1977; Argyris & Schön, 1978). If we view the human intellect as an optimizer, this is indeed an optimizer that can optimize itself, i.e. self-referential optimization (Good, 1966; Schmidhuber, 1987). Such self-referential optimization is enabled by the mutual evolution of task solution and optimizer. On the one hand, the optimizer drives the evolution of solutions, with evolved optimization accelerating the evolution. On the other hand, the optimization of optimizer does not occur in a vacuum: it requires the grounding of concrete tasks to provide the empirical feedback necessary to guide the evolution of the optimizer. This mutually beneficial evolution of both the task solution and the optimizer, driven by grounded task execution, is precisely what allows intelligence to transcend

[†]Equal contribution. [*]Corresponding author. [1]Shenzhen X-Institute [2]Soochow University [3]Shenzhen Loop Area Institute [4]Tsinghua University [5]National University of Singapore. Correspondence to: Han Hao <haoh23@mails.tsinghua.edu.cn>, Liu Yang <yangliu@nus.edu.sg>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

the inherent limitations of its initial state.

This dynamic mutual evolution is elegantly captured by the Escherian metaphor of "Drawing Hands" (Escher, 1948; Hofstadter, 1979). In Escher's famous lithograph, two hands emerge from the flat page to draw one another into existence, where each hand is simultaneously the creator and the creation. Building on this philosophy, we propose *Escher-Loop*, a general agent framework that operationalizes this mutual evolution structure. Escher-Loop maintains a self-referential optimization between two distinct populations: (1) a *Task Agent Population* which directly solves a concrete task, e.g., a population of search algorithms, mathematical proofs, or AI customer service agents, and (2) an *Optimizer Agent Population* which aims to refine task agents. The self-referential nature of our framework lies in that the optimization agent population is recursively applied to optimize itself. In this framework, task-level feedback, particularly the empirical performance scores, serves a dual purpose: it guides the refinement of task agents and simultaneously provides the metrics to evaluate and evolve the optimization agents. This closed-loop architecture ensures the system is constantly sharpened in the crucible of task-level execution. By iteratively optimizing the task and optimizer agents through this grounded feedback loop, Escher-Loop minimizes the reliance on human-designed priors and achieves higher performance ceilings and superior adaptability than systems relying on fixed optimization logic.

Our contributions are summarized as follows:

- **The Escher-Loop Framework.** We formalize the foundational capability of an intelligent agent as the recursive optimization of both the task solution and its own optimization capability, grounded by concrete task feedback. Starting from this perspective, we introduce Escher-Loop, a self-referential optimization framework that enables reciprocity between task agents and optimization agents in a fully closed-loop manner.
- **Dynamic benchmarking without overhead.** To maintain evaluation of optimizers during the continuous evolution of task agents, we propose a dynamic benchmarking mechanism that seamlessly reuses the empirical scores of newly generated task agents as relative win-loss signals to update optimizers' scores. This mechanism leverages the evolution of task agents as an inherent signal to drive the evaluation and refinement of optimizers without additional overhead.
- **Empirical Validation.** Through experiments, we demonstrate that Escher-Loop autonomously discovers sophisticated optimization strategies that outperform static evolutionary baselines, achieving higher performance ceilings.The code is open-sourced at github.com/scaling-group/escher-loop.

2. Escher-Loop: Closed-Loop Self-Referential Optimization

2.1. Overview of the Escher-Loop Framework

To transcend the limitations of human-designed priors and static workflows, we propose *Escher-Loop*, a general framework that operationalizes the mutual evolution of task solutions and optimization capabilities. Escher-Loop maintains a self-referential optimization loop between two distinct populations: a Task Agent Population ($\mathcal{T}$) and an Optimizer Agent Population ($\mathcal{O}$).

Abstractly, within this general framework, a Task Agent is a piece of executable code designed to directly solve downstream tasks, and an Optimizer Agent is essentially another piece of code designed to optimize code. In the specific implementation presented in this paper based on OpenEvolve (Sharma, 2025), we instantiate the optimizer agent as a textual prompt that instructs a Large Language Model (LLM) to perform the optimization, but we highlight that the Escher-Loop framework is general and can be applied to other optimized forms. The framework is initialized with a task evaluation function f, along with initial populations $\mathcal{O}_0$ and $\mathcal{T}_0$. By iteratively running a closed-loop execution, the optimizer drives the evolution of task solutions, while the empirical feedback from grounded task execution simultaneously guides the recursive refinement of the optimizers themselves. Ultimately, the algorithm returns the fully optimized populations $\mathcal{O}$ and $\mathcal{T}$ upon convergence.

2.2. Formalizing the Optimization Step

In our algorithm, both populations are explicitly maintained as sets of tuples containing the agents and their respective empirical scores, denoted as $\mathcal{T} = \{(t_j, s_j^t)\}$ and $\mathcal{O} = \{(o_i, s_i^o)\}$. During the evolutionary process, an optimizer agent o is sampled to transform a selected group of task agents into a new, improved candidate. Formally, we represent this generation process by passing the task agents and their corresponding scores together as tuples:

$$t_{\text{new}} = o((t_1, s_1^t), \ldots, (t_n, s_n^t)) \tag{1}$$

By explicitly providing the task agents alongside their empirical scores, the optimizer can identify successful optimization strategies and steer the generation toward meaningful improvements.

Crucially, because an optimizer agent is essentially a piece of code (instantiated as a program that constructs prompts in our implementation), it is fundamentally no different in data modality from the code-based task agents. This structural equivalence means that the optimizer itself can naturally serve as a target for optimization. Just as an optimizer can refine task code, it can seamlessly rewrite and optimize another optimizer's code (or prompt). This property is exactly

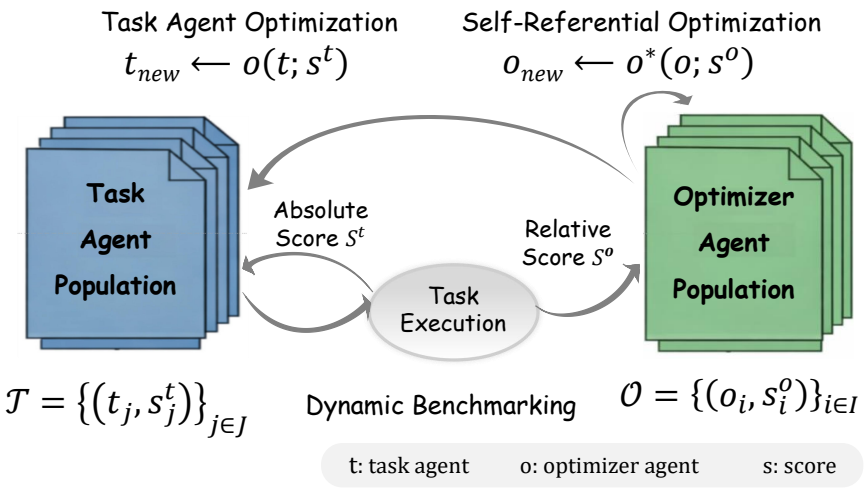

Figure 1. **Sketch of Escher-Loop.** The Escher-Loop maintains a closed loop between task agents and optimizer agents. When sampled optimizers generate new task-solving agents, their execution outcomes are evaluated to obtain absolute task scores s^t. We seamlessly reuse these empirical scores as relative win-loss signals to update the optimizers' scores s^o. This dynamic benchmarking mechanism leverages the evolution of task agents as an inherent signal to drive the evaluation and refinement of optimizers without additional overhead, ultimately yielding the joint evolution of both task performance and optimization capability.

what enables the self-referential nature of our framework, allowing the optimization agent population to be recursively applied to optimize itself. This self-referential optimization step is formulated as:

$$o_{\text{new}} = o((o_1, s_1^o), \ldots, (o_m, s_m^o)) \tag{2}$$

2.3. Dynamic Benchmarking for Optimizers without Overhead

In our framework, evaluating a task agent and maintaining its empirical score s_j^t is straightforward. Because task agents are designed to directly solve concrete problems, their performance can be objectively and easily measured using the provided static task evaluation function f. However, accurately evaluating and updating the score s_i^o for each optimizer agent presents a critical challenge.

Driven by the continuous refinement loop, the task agents are constantly evolving and improving. If we were to evaluate every single optimizer on the most current task agents at every step, the computational cost would be prohibitively high. Indeed, we can only afford to evaluate a subset of optimizer agents at each step. However, this selective evaluation introduces a severe incomparability problem. Because the task agents are constantly being updated, some optimizer agents hold scores derived from optimizing early, naive task agents, while others are evaluated on the most recent, highly refined task agents. The absolute performance improvements achieved on these vastly different generations of task agents cannot be directly compared.

To resolve this discrepancy, we introduce a dynamic benchmarking approach that abandons absolute scoring in favor of relative win-loss outcomes. Instead of measuring absolute score improvements, we apply the sampled subset of optimizers to the same task agents and compare the performance of their newly generated task agents, obtaining pair-wise win-loss signals between the competing optimizers. These win-loss outcomes are integrated into an Elo rating (Elo, 1978) system to maintain a globally consistent and fair evaluation across the entire optimizer population over time.

Notably, our Elo-based dynamic benchmarking not only allows the system to robustly maintain optimizer scores despite the shifting evaluation landscape, but also substantially improves resource allocation without computational overhead. Instead of drawing subsets uniformly at random, we predominantly sample top optimizer agents with the explicit goal of generating better task agents. As a mandatory step, the generated new task agents are evaluated by f to obtain their empirical scores s^t. We seamlessly reuse these exact same scores to determine relative win-loss outcomes and update the participating optimizers' Elo ratings s^o, establishing a more fine-grained ranking specifically among the sampled top-performing optimizers. This elegant dual-use mechanism ensures that we focus on driving the task agent evolution with the most promising optimization agents, while also evaluating and differentiating these optimizers almost "for free".

2.4. The Closed-Loop Mutual Evolution

To synthesize the mechanisms above, Escher-Loop operates as a continuous mutual evolution loop. The process

Algorithm 1 Escher-Loop

Input: task evaluator f; initial populations $\mathcal{O}_0, \mathcal{T}_0$ of optimizer/task agents with scores.
Output: optimized populations $\mathcal{O}, \mathcal{T}$.
$\mathcal{O} \leftarrow \mathcal{O}_0$; $\mathcal{T} \leftarrow \mathcal{T}_0$.
while not converged **do**
  Sample populations:
  $\{(o_i, s_i^o)\}_{i \in I} \leftarrow \textsc{Sample}(\mathcal{O})$;$\{(t_j, s_j^t)\}_{j \in J} \leftarrow \textsc{Sample}(\mathcal{T})$.
  Optimize and score task agents:
  for $i \in I$ **do**
    $\hat{t}_i \leftarrow o_i(\{(t_j, s_j^t)\}_{j \in J})$.
    $\hat{s}_i^t \leftarrow f(\hat{t}_i)$.
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\hat{t}_i, \hat{s}_i^t)\}$.
  end for
  Update optimizer scores from task feedback:
  $W \in \{-1, 0, 1\}^{|I| \times |I|} \leftarrow \textsc{PairwiseCompetition}(\{\hat{s}_i^t\}_{i \in I})$.
  $\{s_i^o\}_{i \in I} \leftarrow \textsc{EloUpdate}(W, \{s_i^o\}_{i \in I})$.
  Self-referential optimization:
  $(o^*, -), \{(o_i, s_i^o)\}_{i \in \hat{I}} \leftarrow \textsc{Sample}(\mathcal{O})$.
  $\hat{o} \leftarrow o^*(\{(o_i, s_i^o)\}_{i \in \hat{I}})$.
  $\hat{s}^o \leftarrow \textsc{Initialize}(\{s_i^o\}_{i \in \hat{I}})$.
  $\mathcal{O} \leftarrow \mathcal{O} \cup \{(\hat{o}, \hat{s}^o)\}$.
end while
return $\mathcal{O}, \mathcal{T}$.

is initialized with optimizer and task populations, $\mathcal{O}_0$ and $\mathcal{T}_0$, together with the task evaluation function f, and then alternates between improving task agents and improving the optimizer population that produces them.

At each iteration, the algorithm samples subsets from both populations using a unified rank-based Softmax mechanism. By relying on relative ranks rather than absolute score magnitudes, the sampling remains robust under heterogeneous and shifting score scales, while the temperature parameter controls the exploration–exploitation trade-off.

The sampled optimizers generate new task candidates from the sampled task agents. These candidates are immediately evaluated by f and integrated into $\mathcal{T}$. The same empirical task scores then serve a second role: they provide win-loss evidence for updating the participating optimizers' Elo ratings through the dynamic benchmarking procedure, thereby linking optimizer selection to grounded task feedback rather than to a separate evaluator.

The loop then executes the self-referential step. A lead optimizer o^* rewrites a sampled subset of the optimizer population to synthesize a new optimizer candidate $\hat{o}$, whose initial score is inherited from its predecessors before it is added back into $\mathcal{O}$. This final step makes the optimizer population itself an evolving object rather than a fixed search heuristic.

This cycle of rank-based sampling, task generation, dual-purpose evaluation, and self-referential optimizer evolution continuously sharpens both concrete task agents and optimization capabilities. The overall procedure is summarized in Algorithm 1.

3. Results and Analysis

We evaluate Escher-Loop along two complementary axes: (1) whether closed-loop self-referential optimization improves task-level search under a controlled compute budget compared to static baselines, and (2) how the underlying mechanisms, specifically the prioritization driven by dynamic benchmarking and the autonomous emergence of sophisticated optimization strategies, enable this superior performance.

Experimental Protocol We consider three geometric optimization landscapes: Kissing Number (KN), Circle Packing (CP), and Heilbronn Triangle (HT). To ensure a fair cost comparison, all methods are evaluated under a strict budget of 10M equivalent tokens, a metric that normalizes the combined cost of input and output tokens into a single output-token equivalent based on relative API pricing. Performance is reported using normalized task scores for the Task Agent Population and Elo ratings for the Optimizer Agent Population. Detailed experimental setups can be found in Appendix A.

3.1. Task Performance

We employ OpenEvolve (Sharma, 2025), an open-source reproduction of AlphaEvolve, as the representative handcrafted baseline optimizer. Escher-Loop is compared against this static baseline across the three optimization landscapes. To ensure a fair comparison of computational cost, the evolving trajectories are evaluated by plotting the best-so-far normalized task score as a function of token consumption on the horizontal axis.

Results As illustrated in Figure 2, Escher-Loop consistently improves the optimization ceiling on the harder KN and CP landscapes, where the handcrafted baseline often enters long plateaus. On HT, the baseline reaches high scores quickly, but Escher-Loop still attains the highest peak score among our runs.

Analysis These results suggest that fixed handcrafted heuristics can be efficient on simpler landscapes, but are less robust on tasks where the useful search strategy changes over time. Escher-Loop benefits from evolving the optimization process itself, allowing it to escape suboptimal regimes and reach higher task scores.

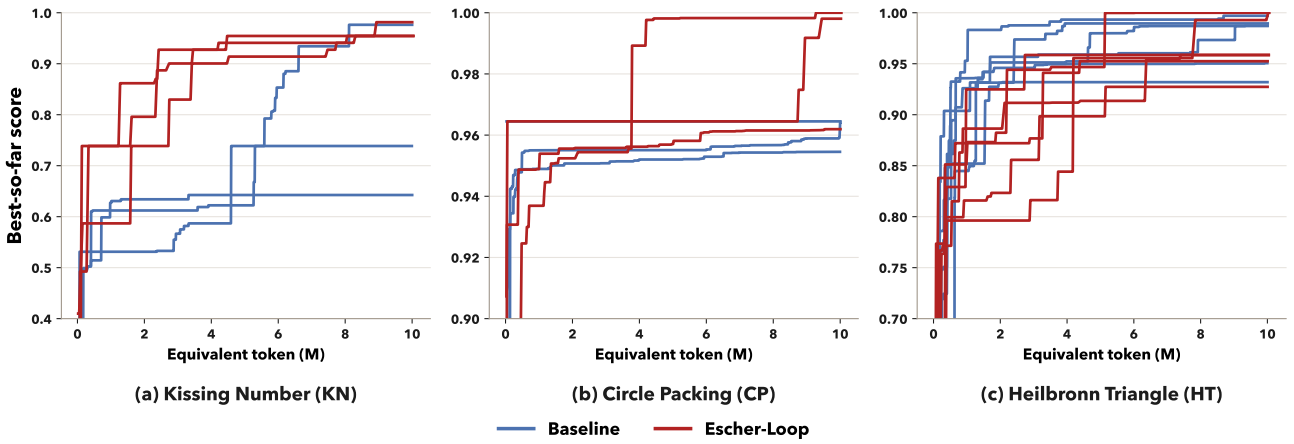

Figure 2. Comparison of best-so-far task performance between the handcrafted baseline optimizer and Escher-Loop across three optimization landscapes: Kissing Number (KN), Circle Packing (CP), and Heilbronn Triangle (HT). Each curve represents one independent run, and the reported score is the best-so-far normalized task score as a function of cumulative equivalent token consumption.

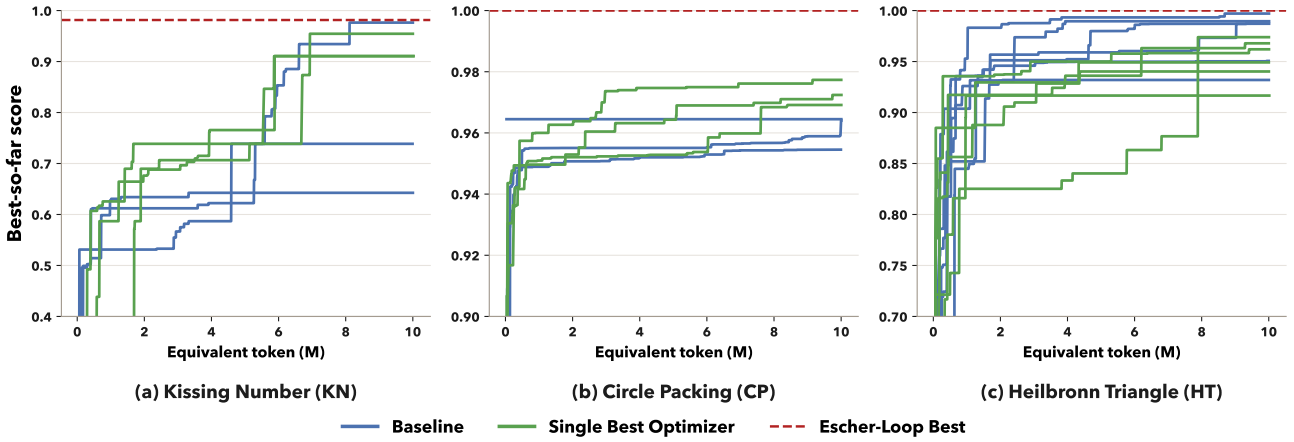

Figure 3. Comparison of best-so-far normalized task performance between the handcrafted baseline optimizer and the single best evolved optimizer across three optimization landscapes: Kissing Number (KN), Circle Packing (CP), and Heilbronn Triangle (HT). Each curve represents one independent run, while the horizontal dashed line indicates the best performance achieved by Escher-Loop.

3.2. The Necessity of Evolving Optimizer Population

To evaluate the intrinsic capability of the evolved optimizers, we isolate the highest-scoring agent from the Escher-Loop evolution process, termed the *Single Best Optimizer (SBO)*. By substituting the handcrafted optimizer in OpenEvolve with this SBO, we directly compare their isolated trajectories. Note that the best performance developed by Escher-Loop is denoted by a horizontal dashed upper bound.

Results As shown in Figure 3, the SBO transfers useful search behavior on KN and CP, often outperforming the handcrafted baseline. However, it underperforms the baseline on HT, and across all three landscapes it never exceeds the peak score achieved by the full Escher-Loop framework.

Analysis The SBO result shows that self-referential evolution can produce strong optimizer agents, but also that a single frozen optimizer does not reproduce the full system. The advantage of Escher-Loop comes from maintaining an evolving optimizer population whose search policy can keep adapting as the task population changes.

3.3. Mechanism Ablations

The previous results show that Escher-Loop improves over both the handcrafted baseline and a single transferred optimizer. We next test which parts of the closed-loop mechanism are responsible for this improvement. Since Escher-Loop exhibits the clearest gains on Kissing Number (KN) and Circle Packing (CP), we conduct mechanism ablations on these two tasks.

We consider three ablated variants. ***Random Elo Update*** preserves the rating interface but breaks its semantic connection to task feedback: after task execution, optimizer ratings are randomly perturbed rather than updated according to the observed relative task outcomes. This approximates an optimizer population whose selection pressure is effectively random. ***Fixed Elo = 1200*** removes relative optimizer differentiation by keeping all optimizer ratings fixed, so optimizer agents are no longer ranked by empirical evidence from task improvement. ***Static-50 Optimizer Pool*** preserves optimizer diversity but removes self-referential optimizer evolution: we initialize a fixed pool of 50 manually selected optimizer agents, matching the optimizer population cap used in Escher-Loop, and use this pool to optimize task agents without evolving the optimizer population itself. This ablation tests whether population diversity alone explains the gains.

Each ablation is repeated three times with the same 10M equivalent-token budget used in the main Baseline and Escher-Loop comparisons. We summarize performance using two metrics, reporting the mean across runs. *Best@10M* is the best normalized task score reached within the budget. *AUC@10M* is the area under the best-so-far task-score curve over the same budget, normalized by the budget:

$$\text{AUC@10M} = \frac{1}{10M} \int_0^{10M} \text{best_so_far}(t)\, dt. \quad (3)$$

In practice, this integral is computed from the recorded discrete trajectory as a step-function area. Thus, Best@10M measures the final achieved solution quality, while AUC@10M measures search efficiency throughout the run. Since several tasks exhibit discrete performance plateaus, run-to-run variability in final best score can be visually misleading; we therefore report variability through AUC standard deviation, which captures differences in when high-quality solutions are discovered.

Table 1. Mechanism ablations on Kissing Number. Each ablation is repeated three times. Best@10M and AUC@10M report mean values under the 10M-token budget, following the same repeated-run protocol as the main comparisons; AUC Std summarizes run-to-run stability.

Method	Best@10M ↑	AUC@10M ↑	AUC Std ↓
Baseline	0.786	0.674	0.067
Single Best Optimizer	0.925	0.738	0.048
Static-50 Optimizer Pool	0.568	0.408	0.181
Fixed Elo = 1200	0.811	0.772	0.074
Random Elo Update	0.739	0.721	0.016
Escher-Loop (ours)	**0.963**	**0.876**	0.032

Analysis The ablations indicate that Escher-Loop's gains do not arise merely from using more optimizer agents. On KN, the Static-50 pool has the largest variance and substantially lower AUC, showing that a fixed diverse optimizer pool can occasionally discover useful search directions but fails to provide stable optimization pressure. Fixed Elo and Random Elo preserve the appearance of an optimizer population, yet both remove the grounded relative selection signal that connects optimizer quality to task feedback; they reach the common KN plateau but fall below Escher-Loop in both final performance and search efficiency. On CP, the baseline is already strong and all methods are closer in absolute score; Static-50 attains a slightly higher mean peak score, while Escher-Loop remains effectively tied in AUC and substantially stronger on KN. Together, these results support the central claim that optimizer diversity alone is insufficient: it must be coupled with grounded, dynamic selection and self-referential optimizer evolution.

Table 2. Mechanism ablations on Circle Packing. Each ablation is repeated three times. Best@10M and AUC@10M report mean values under the 10M-token budget, following the same repeated-run protocol as the main comparisons; AUC Std summarizes run-to-run stability.

Method	Best@10M ↑	AUC@10M ↑	AUC Std ↓
Baseline	0.961	0.956	0.008
Single Best Optimizer	0.973	0.962	0.007
Static-50 Optimizer Pool	**0.988**	**0.966**	0.009
Fixed Elo = 1200	0.969	0.959	0.003
Random Elo Update	0.964	0.961	0.001
Escher-Loop (ours)	0.987	**0.966**	0.009

3.4. Qualitative Analysis of Emergent Optimization Strategies

The continuous improvements in task performance observed in our experiments emerge from the synergistic mutual evolution between the Task Agent and Optimizer Agent populations. As the optimizers generate stronger task agents, the shifting evaluation landscape in turn drives the optimizers to evolve more capable optimization strategies. As anticipated by our closed-loop design, the optimizers do not merely accumulate code complexity; rather, they develop increasingly sophisticated optimization strategies. As illustrated by the representative code segments in Figure 4, the evolved optimizer transcends basic LLM prompting. It shifts away from the fixed logic of handcrafted baselines and develops strategies that can process rich evolutionary feedback to make informed search decisions.

We observe that this capacity for autonomous evolution naturally extends across all three problem domains considered in this work. Specifically, they discover intuitive and highly effective search strategies, such as reasoning over historical attempts to avoid repeated failures, and extracting beneficial logic from diverse candidate programs to combine them. This consistent ability to dynamically synthesize and deploy context-appropriate optimization strategies demonstrates the effectiveness of our self-referential framework.

```diff
@@ -169,18 +177,18 @@
         if previous_programs:
             prev_metrics = previous_programs[-1].get("metrics", {})
             prev_fitness = get_fitness_score(prev_metrics, feature_dimensions)
-            if current_fitness > prev_fitness + 1e-6:
-                improvement_areas.append(
-                    self.template_manager.get_fragment(
-                        "fitness_improved", prev=prev_fitness, current=current_fitness
-                    )
-                )
-            elif current_fitness < prev_fitness - 1e-6:
-                improvement_areas.append(
-                    self.template_manager.get_fragment(
-                        "fitness_declined", prev=prev_fitness, current=current_fitness
-                    )
-                )
+
+            if current_fitness < prev_fitness - 0.5:
+                improvement_areas.append("REGRESSION RECOVERY: Identify 'Logic Regression' and REVERT.")
+            elif abs(current_fitness - prev_fitness) < 0.0001:
+                improvement_areas.append("HEURISTIC PLATEAU: Perform a 'Paradigm Shift' in logic.")
+
-        if not improvement_areas:
-            improvement_areas.append(self.template_manager.get_fragment("no_specific_guidance"))
+        for dim in feature_dimensions:
+            val = metrics.get(dim, 0)
+            if isinstance(val, (int, float)) and val < 0.0001:
+                if dim == "semantic_diversity":
+                    improvement_areas.append("CRITICAL DIMENSIONAL GAP: Implement 'Black Swan' event.")
+                else:
+                    improvement_areas.append(f"EXPLORATION DEFICIT: '{dim}' is {val:.4f}. Prioritize.")
```

(a) **Diagnostic feedback mechanism** The optimizer replaces generic fitness-improvement messages with explicit recovery and plateau signals, such as regression recovery and low-diversity warnings.

```diff
@@ -37,6 +41,35 @@
         if not hasattr(logger, "_prompt_sampler_logged"):
             logger.info("Initialized prompt sampler")
             logger._prompt_sampler_logged = True

+    def _apply_config_overrides(self) -> None:
+        """Dynamically tune search entropy from stagnation and round state."""
+        stagnation = getattr(self, "_last_stagnation_count", 0)
+        round_num = getattr(self, "_current_round", 0)
+        if not hasattr(self, "_prev_lengths"):
+            self._prev_lengths = []
+        prev_lens = self._prev_lengths
+        curr_len = prev_lens[-1] if prev_lens else 0
+        is_rigid = len(prev_lens) > 3 and all(l == curr_len for l in prev_lens[-3:]) and stagnation > 0
+        is_bloating = len(prev_lens) > 2 and curr_len > prev_lens[-2] and stagnation > 0
+
+        if stagnation >= 4 or is_rigid:
+            self.config.num_diverse_programs = 6
+            self.config.num_top_programs = 0
+            self.config.use_meta_prompting = True
+            self.config.suggest_simplification_after_chars = 350 if is_bloating else 600
+        elif stagnation >= 2:
+            self.config.num_diverse_programs = 3
+            self.config.num_top_programs = 2
+            self.config.use_meta_prompting = True
+        elif round_num < 15:
+            self.config.num_diverse_programs = 4
+            self.config.num_top_programs = 1
+            self.config.use_meta_prompting = True
+        else:
+            self.config.num_diverse_programs = 1
+            self.config.num_top_programs = 4
+            self.config.use_template_stochasticity = False
+
     def set_templates(
         self, system_template: Optional[str] = None, user_template: Optional[str] = None
     ) -> None:
```

(b) **Adaptive search control** The optimizer adjusts exploration pressure according to stagnation, code rigidity, bloat, and evolution stage, shifting between high-diversity exploration and elite-focused refinement.

```diff
@@ -85,10 +91,15 @@
         if self.config.use_template_stochasticity:
             user_template = self._apply_template_variations(user_template)

+        system_msg = self._get_system_message(
+            improvement_text=improvement_summary,
+            round_num=evolution_round,
+        )
         return {
-            "system": self._get_system_message(),
+            "system": system_msg,
             "user": safe_format(user_template, **payload),
         }

-    def _get_system_message(self) -> str:
+    def _get_system_message(self, improvement_text: str = "", round_num: int = 0) -> str:
         if self.system_template_override:
             return self.template_manager.get_template(self.system_template_override)
@@ -136,14 +241,28 @@
-        system_message = self.config.system_message
-        if system_message in self.template_manager.templates:
-            return self.template_manager.get_template(system_message)
-        return system_message
+        base_system = self.config.system_message
+        if base_system in self.template_manager.templates:
+            base_system = self.template_manager.get_template(base_system)
+
+        mode = "EXPLORATION"
+        if "REGRESSION" in improvement_text:
+            mode = "RECOVERY"
+        elif "STAGNATION" in improvement_text:
+            mode = "MACRO-MUTATION"
+        elif "PROGRESS" in improvement_text:
+            mode = "EXPLOITATION"
+
+        directives = {
+            "RECOVERY": "Revert to the strongest reference program.",
+            "MACRO-MUTATION": "Change the algorithmic family or data structure.",
+            "EXPLOITATION": "Preserve the core algorithm and refine locally.",
+            "EXPLORATION": "Prioritize correctness and robust baselines.",
+        }
+        return (
+            f"{base_system}\n\n"
+            f"### EVOLUTIONARY PROTOCOL: {mode}\n"
+            f"Directive: {directives[mode]}\n"
+            "Reasoning: contrast, attribute failure, then update."
+        )
```

(c) **Stage-aware system prompting** The optimizer selects different prompt protocols for exploration, regression recovery, macro-mutation, and exploitation, making the system message conditional on recent optimization signals.

```diff
@@ -199,29 +313,30 @@
     def _format_evolution_history(
         self,
         previous_programs: List[Dict[str, Any]],
         top_programs: List[Dict[str, Any]],
         inspirations: List[Dict[str, Any]],
         language: str,
         feature_dimensions: Optional[List[str]] = None,
     ) -> str:
-        previous_attempts = self._render_previous_attempts(...)
-        top_sections = self._render_reference_programs(...)
+        def metric_delta(program: Dict[str, Any]) -> str:
+            parent = program.get("metadata", {}).get("parent_metrics", {})
+            return "\n".join(
+                f"{k}: {v:.4f} {self._get_comparison_arrow(v, parent.get(k, v))}"
+                for k, v in program.get("metrics", {}).items()
+                if isinstance(v, (int, float))
+            )
+
+        previous_attempts = "\n".join(
+            metric_delta(p) for p in previous_programs[-4:]
+        )
+        elite = top_programs[: self.config.num_top_programs]
+        diverse = sorted(
+            top_programs[self.config.num_top_programs :],
+            key=lambda p: p.get("metrics", {}).get("semantic_diversity", 0),
+            reverse=True,
+        )[: self.config.num_diverse_programs]
+        top_sections = "\n".join(
+            f"{get_fitness_score(p.get('metrics', {}), feature_dimensions or []):.4f}: "
+            f"{self._extract_unique_features(p)}\n{p.get('code', '')}"
+            for p in elite + diverse
+        )
@@ -307,14 +386,10 @@
-            program_snippet="",
-            language=language,
-        )
+    def _get_comparison_arrow(self, current: float, previous: float) -> str:
+        if current > previous: return "↑"
+        if current < previous: return "↓"
+        return "→"
@@ -447,6 +479,10 @@
+    def _extract_unique_features(self, program: Dict[str, Any]) -> str:
+        code = program.get("code", "").lower()
+        tags = ["Vectorized"] if "np." in code or "numpy" in code else []
+        if "@lru_cache" in code or "memo =" in code: tags.append("Memoized")
+        return " | ".join(tags) or "Baseline"
```

(d) **Reference-program mining** The optimizer expands evolution history into structured references by tracking metric deltas, selecting elite and diverse programs, and extracting reusable architectural features from prior candidates.

Figure 4. **Representative code segments from evolved optimizer agents.** The four examples illustrate how self-referential optimization reshapes the optimizer's prompt-construction logic by introducing diagnostic feedback, adaptive search control, stage-specific prompting, and reference-program mining. Together, these snippets provide qualitative evidence that the optimizer population develops increasingly sophisticated strategies for interpreting task feedback and structuring future search, rather than relying solely on a fixed handcrafted prompt.

4. Related Works

Language Models as Optimizers Large language models (LLMs) are naturally suited for gradient-free optimization, ranging from optimizing natural language prompts (Yang et al., 2024a; Fernando et al., 2024) to evolving agent scaffolds like DAG configurations (Hu et al., 2025; Liu et al., 2026a) and optimizing initialization seed (Tan et al., 2026). Another prominent direction treats program search as an evolutionary cycle (Schmidhuber, 1987), where frameworks like FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) use LLMs as code-level mutation operators for algorithmic discovery. Subsequent extensions, whether incorporating heuristic modifications like memory archives, such as OpenEvolve (Sharma, 2025), ThetaEvolve (Wang et al., 2025), ShinkaEvolve (Lange et al., 2026), EvoX (Liu et al., 2026b), and TTT-Discover (Yuksekgonul et al., 2026), designing adaptive search controllers, such as AdaEvolve (Cemri et al., 2026), or scaling these evaluation loops like SimpleTES (Ye et al., 2026), remain entangled with human-engineered heuristics and confined to static workflows.

Recursive Self-Improvement (RSI) While using LLMs as static optimizers yields strong performance, the pursuit of open-ended growth requires systems that can improve their own optimization mechanisms. Pioneered by Good (1966) and later formalized by the Gödel Machine (Schmidhuber, 2003), the concept of self-referential systems has been fundamentally explored in neural networks (Kirsch & Schmidhuber, 2022). Recently, this pursuit has transitioned into empirical LLM-driven code-level self-modification. Frameworks such as STOP (Zelikman et al., 2024), Gödel Agent (Yin et al., 2025), and SICA (Robeyns et al., 2025) eliminate the boundary between target and meta-agents, enabling systems to dynamically diagnose and rewrite their own codebases. To overcome the limitations of greedy single-path updates, DGM (Zhang et al., 2026a) introduces a growing tree of historical agent variants, while HGM (Wang et al., 2026) utilizes clade-based meta-productivity to balance exploration. Further abstracting this capability, Hyperagents (Zhang et al., 2026b) unifies task and meta-level logic into an editable program library, enabling cross-domain metacognitive optimization.

Population Evolution To overcome the limited exploration and overfitting of isolated individual RSI, recent works such as GEA (Weng et al., 2026) and TerraLingua (Paolo et al., 2026) expand the evolutionary unit to populations by introducing lateral experience transfer and environmental dynamics. Pushing this toward full autonomy, CORAL (Qu et al., 2026) enables asynchronous agents to co-evolve via shared persistent memory without fixed heuristics. More closely related to our closed-loop approach, frameworks like CoEvolve (Yang et al., 2026) and Paso Doble (Zhang et al., 2026c) introduce *mutual evolution*, where the optimizer continuously synthesizes targeted adversarial tasks to co-evolve alongside the policy.

5. Conclusion

In this work, we shift the focus of autonomous agents from task-level solutions to *optimization ability* itself, and introduce Escher-Loop, a closed-loop, self-referential framework in which optimization becomes the object of evolution. Our results suggest that optimization ability can be treated as a comparable and continuously improvable property, rather than a fixed mechanism prescribed by human design.

Beyond the empirical results, our findings further illuminate how such improvement becomes possible. The closed-loop coupling between optimizer agents and a dynamically evolving task population creates a continually shifting evaluation landscape, in which optimization strategies are repeatedly tested, compared, and refined under relative feedback. This interaction induces a form of pressure analogous to natural selection, enabling optimization strategies to adapt and accumulate improvements over time without explicit supervision or retraining. As a result, self-referential improvement emerges not as a predefined objective, but as a natural outcome of the synergistic mutual evolution within this closed-loop system.

More broadly, Escher-Loop highlights a shift in objective from a single task solution to the iterative optimization process itself. While our current exploration demonstrates self-referential improvement within individual problem spaces, it opens the door to an even more ambitious vision: *cross-domain multi-task optimization*. In such future systems, generalization would no longer be achieved solely through improved task-specific performance, but through a unified, transferable optimization ability that governs how solutions are updated across diverse tasks and environments. We believe this perspective opens a path toward more autonomous and scalable forms of intelligence, in which continual improvement emerges as an intrinsic property of the system. We leave the full validation of this unified view and the achievement of true cross-domain transferability to future work.

Acknowledgments

Ziyang Liu, Xinyan Guo, and Xuchen Wei were supported by Shenzhen X-Institute through its Pioneering Investigator Program. Liu Yang acknowledges support from the National Research Foundation, Singapore, under the NRF fellowship (Project No. NRF-NRFF17-2025-0006).

References

Anthropic. Claude 3.7 sonnet and claude code, February 2025. URL https://www.anthropic.com/news/claude-3-7-sonnet.

Argyris, C. Double loop learning in organizations. *Harvard business review*, 55(5):115–125, 1977.

Argyris, C. and Schön, D. A. *Organizational Learning: A Theory of Action Perspective*. Addison-Wesley, Reading, MA, 1978.

Cemri, M., Agrawal, S., Gupta, A., Liu, S., Cheng, A., Mang, Q., Naren, A., Erdogan, L. E., Sen, K., Zaharia, M., Dimakis, A., and Stoica, I. Adaevolve: Adaptive llm driven zeroth-order optimization, 2026. URL https://arxiv.org/abs/2602.20133.

Elo, A. *The rating of chessplayers, past and present*. Arco Pub., 1978. ISBN 978-0-668-04721-0. URL https://books.google.com.sg/books?id=8pMnAQAAMAAJ.

Escher, M. C. Drawing hands. Lithograph, 1948.

Fernando, C., Banarse, D. S., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution. In Salakhutdinov, R., Kolter, Z., Heller, K., Weller, A., Oliver, N., Scarlett, J., and Berkenkamp, F. (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 13481–13544. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/fernando24a.html.

Flavell, J. H. Metacognition and cognitive monitoring: A new area of cognitive-developmental inquiry. *American Psychologist*, 34(10):906–911, 1979. doi:10.1037/0003-066X.34.10.906.

Good, I. J. Speculations concerning the first ultraintelligent machine. In *Advances in computers*, volume 6, pp. 31–88. Elsevier, 1966.

Google DeepMind. Gemini 3 flash model card. Technical report, Google, December 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf.

Hofstadter, D. R. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, New York, 1979.

Hu, S., Lu, C., and Clune, J. Automated design of agentic systems. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=t9U3LW7JVX.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Kirsch, L. and Schmidhuber, J. Eliminating meta optimization through self-referential meta learning, 2022. URL https://arxiv.org/abs/2212.14392.

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

Liu, J., Jiang, Y., Zhang, G., Zhang, Z., Chang, H., Yin, Z., Ren, Q., and Yan, J. Todoevolve: Learning to architect agent planning systems, 2026a. URL https://arxiv.org/abs/2602.07839.

Liu, S., Agarwal, S., Maheswaran, M., Cemri, M., Li, Z., Mang, Q., Naren, A., Boneh, E., Cheng, A., Pan, M. Z., Du, A., Keutzer, K., Cheung, A., Dimakis, A. G., Sen, K., Zaharia, M., and Stoica, I. Evox: Meta-evolution for automated discovery, 2026b. URL https://arxiv.org/abs/2602.23413.

Luo, J., Zhang, W., Yuan, Y., Zhao, Y., Yang, J., Gu, Y., Wu, B., Chen, B., Qiao, Z., Long, Q., Tu, R., Luo, X., Ju, W., Xiao, Z., Wang, Y., Xiao, M., Liu, C., Yuan, J., Zhang, S., Jin, Y., Zhang, F., Wu, X., Zhao, H., Tao, D., Yu, P. S., and Zhang, M. Large language model agent: A survey on methodology, applications and challenges, 2025. URL https://arxiv.org/abs/2503.21460.

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

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Paolo, G., Warner, J., Shahrzad, H., Hodjat, B., Miikkulainen, R., and Meyerson, E. TerraLingua: Emergence and analysis of open-endedness in llm ecologies, 2026. URL https://arxiv.org/abs/2603.16910.

Qu, A., Zheng, H., Zhou, Z., Yan, Y., Tang, Y., Ong, S. Y., Hong, F., Zhou, K., Jiang, C., Kong, M., Zhu, J., Jiang, X., Li, S., Wu, C., Low, B. K. H., Zhao, J., and Liang, P. P.

Coral: Towards autonomous multi-agent evolution for open-ended discovery, 2026. URL https://arxiv.org/abs/2604.01658.

Robeyns, M., Szummer, M., and Aitchison, L. A self-improving coding agent. In *Scaling Self-Improving Foundation Models without Human Supervision*, 2025. URL https://openreview.net/forum?id=rShJCyLsOr.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J., Ellenberg, J. S., Wang, P., Fawzi, O., et al. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024.

Schmidhuber, J. Evolutionary principles in self-referential learning, or on learning how to learn: The meta-meta-... hook. Diploma thesis, Institut für Informatik, Technische Universität München, Germany, 1987. URL https://people.idsia.ch/~juergen/diploma1987ocr.pdf.

Schmidhuber, J. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. Technical Report IDSIA-19-03, arXiv:cs.LO/0309048, IDSIA, Manno-Lugano, Switzerland, 2003. Revised 2006.

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

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: language agents with verbal reinforcement learning. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/1b44b878bb782e6954cd888628510e90-Paper-Conference.pdf.

Sutton, R. S. The bitter lesson, Mar 2019. URL http://www.incompleteideas.net/IncIdeas/BitterLesson.html. Accessed: 2026-02-07.

Tan, D. Y. Y., Chin, K., and Zhang, J. AgentGA: Evolving code solutions in agent-seed space, 2026. URL https://arxiv.org/abs/2604.14655.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

Wang, W., Piękos, P., Li, N., Laakom, F., Chen, Y., Ostaszewski, M., Zhuge, M., and Schmidhuber, J. Huxley-Gödel Machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=T0EiEuhOOL.

Wang, Y., Su, S.-R., Zeng, Z., Xu, E., Ren, L., Yang, X., Huang, Z., He, X., Ma, L., Peng, B., Cheng, H., He, P., Chen, W., Wang, S., Du, S. S., and Shen, Y. Thetaevolve: Test-time learning on open problems, 2025. URL https://arxiv.org/abs/2511.23473.

Weng, Z., Antoniades, A., Nathani, D., Zhang, Z., Pu, X., and Wang, X. E. Group-Evolving Agents: Open-ended self-improvement via experience sharing, 2026. URL https://arxiv.org/abs/2602.04837.

Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=Bb4VGOWELI.

Yang, H., Yue, S., and He, Y. Auto-GPT for online decision making: Benchmarks and additional opinions, 2023. URL https://arxiv.org/abs/2306.02224.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: agent-computer interfaces enable automated software engineering. In Globerson, A., Mackey, L., Belgrave, D., Fan, A., Paquet, U., Tomczak, J., and Zhang, C. (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024b. doi:10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

Yang, S., Ma, Z., Huang, T., Hu, Y., Wang, Y., and Chu, X. CoEvolve: Training llm agents via agent-data mutual evolution, 2026. URL https://arxiv.org/abs/2604.15840.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Ye, H., Lin, H., Tang, J., Luo, Y., Yang, C., Su, C., Thapa, R., Yang, R., Liu, R., Li, Z., Gao, C., Ding, D., He, G., Zhang, M., Sun, L., Wang, W., Zhong, Y., Shen, Z., He, D., Ma, J., Ermon, S., Li, T., Chu, X., Zou, J., and Xu, Y. Evaluation-driven scaling for scientific discovery, 2026. URL https://arxiv.org/abs/2604.19341.

Yin, X., Wang, X., Pan, L., Lin, L., Wan, X., and Wang, W. Y. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 27890–27913, 2025.

Yuksekgonul, M., Koceja, D., Li, X., Bianchi, F., McCaleb, J., Wang, X., Kautz, J., Choi, Y., Zou, J., Guestrin, C., and Sun, Y. Learning to discover at test time, 2026. URL https://arxiv.org/abs/2601.16175.

Zelikman, E., Lorch, E., Mackey, L., and Kalai, A. T. Self-taught optimizer (STOP): Recursively self-improving code generation. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=46Zgqo4QIU.

Zhang, J., Hu, S., Lu, C., Lange, R. T., and Clune, J. Darwin Gödel Machine: Open-ended evolution of self-improving agents. In *The Fourteenth International Conference on Learning Representations*, 2026a. URL https://openreview.net/forum?id=pUpzQZTvGY.

Zhang, J., Zhao, B., Yang, W., Foerster, J., Clune, J., Jiang, M., Devlin, S., and Shavrina, T. Hyperagents, 2026b. URL https://arxiv.org/abs/2603.19461.

Zhang, Z., Huang, C., Li, A. O., and Cardie, C. Better llm reasoning via dual-play, 2026c. URL https://arxiv.org/abs/2511.11881.

A. Implementation Details and Experimental Protocol

This appendix provides implementation details that support reproducibility and clarify the experimental protocol underlying the main results in Section 3. Specifically, we summarize the compute normalization used in matched-budget comparisons, the model-routing and generation configurations, the prompt interface exposed to optimizer agents, and the archive and population settings used in closed-loop evolution.

A.1. Compute Normalization via Equivalent Tokens

All principal comparisons are conducted under a matched *equivalent-token* budget. Because, in the API setting used in our experiments, input tokens are priced at one quarter of output tokens, we normalize total token consumption into a unified output-equivalent measure:

$$T_{\text{eq}} = T_{\text{out}} + 0.25\, T_{\text{in}}, \tag{4}$$

where T_{in} and T_{out} denote the consumed input and output tokens, respectively. This normalization ensures that performance differences cannot be attributed merely to unequal expenditure of inference-time compute. Unless otherwise stated, the matched-budget comparisons in the main text are reported under a budget of 10M equivalent tokens per task.

A.2. Model Ensemble and Generation Settings

To balance reasoning quality, throughput, and latency, we employ a hybrid Gemini 3 Flash (Google DeepMind, 2025) ensemble. The generation and routing configuration is summarized below.

- **Model allocation.** We route 80% of requests to `gemini-3-flash-preview` with `thinkingLevel` set to `low`, while the remaining 20% use the model's default dynamic thinking configuration. This asymmetric allocation provides a practical trade-off between throughput, cost, and reasoning depth.
- **Temperature control.** We use a two-level temperature scheme.
  1. *Generation temperature.* During optimizer evolution, we use a generation temperature of $T = 1.0$ to encourage exploratory variation in candidate proposals. During downstream task optimization, we reduce this temperature to $T = 0.7$ to improve stability and reduce unproductive variance.
  2. *Archive sampling temperature.* The MAP-Elites archive (Mouret & Clune, 2015) uses rank-based Softmax sampling, $P(i) \propto \exp(-\text{rank}_i/T)$, with role-specific temperatures:
     - **Matchmaking ($T = 1.2$):** encourages moderate exploration when selecting opponents of comparable Elo level;
     - **Mentoring ($T = 0.5$):** emphasizes stronger optimizer agents as teachers, thereby reducing entropy during guidance;
     - **Checkpoint selection ($T = 1.2$):** balances exploitation of high-quality archive states against exploration of historically diverse ones.
- **Sequence length.** The `max_tokens` limit is fixed at 60,000 across all phases.
- **Request management.** We use a timeout of 1,200 s during optimizer evolution and 600–800 s during downstream task optimization, with at most three retries per request.

A.3. Rank-Based Population Sampling

Whenever Escher-Loop samples from either the task-agent or optimizer-agent population, it uses rank-based Softmax sampling. Given a candidate set $\mathcal{C}$, each element $i \in \mathcal{C}$ is assigned a rank $\text{rank}(i)$ according to its score, where better candidates receive smaller ranks. The sampling probability is

$$P(i) = \frac{\exp\left(-\text{rank}(i)/\tau\right)}{\sum_{j \in \mathcal{C}} \exp\left(-\text{rank}(j)/\tau\right)}, \tag{5}$$

where τ controls the exploration–exploitation trade-off. This rank-based form avoids relying on absolute score magnitudes, which may be incomparable across evolving task-agent and optimizer-agent populations.

A.4. Optimizer Evolution and MAP-Elites Configuration

The closed-loop system maintains an archive-based optimizer population with the following settings:

- **Population control.** The optimizer population size is set to 50, and checkpoints are saved every 20 iterations.
- **Elo rating.** Optimizer agents are initialized with Elo 1200.0 and updated using a K-factor of 32.0.
- **Archive usage.** The optimizer population is maintained in a MAP-Elites-style archive, where both fitness and behavioral diversity are preserved. This archive is used not only for selection, but also as contextual input to subsequent optimization steps through elite references, diverse inspirations, and historical checkpoints.
- **Dynamic benchmarking.** Optimizer quality is assessed through repeated relative comparisons under

evolving task populations rather than through a single static benchmark. Elo is therefore used as a comparative ranking signal for optimizer agents, while downstream task metrics remain the ground-truth signal for task-program quality.

B. Best Evolved Task Programs

This appendix presents the best-performing task agents discovered by Escher-Loop on the three optimization landscapes studied in the paper. For each task, we first restate the formal problem definition, including its search domain, task-specific parameters, and evaluation rule, and then show the corresponding program. Throughout this appendix, the normalized score is computed as

$$s_{\text{norm}} = \frac{s_{\text{raw}}}{s_{\text{ref}}}, \tag{6}$$

where s_{raw} is the task-native evaluator output and s_{ref} is the reference value used for normalization in the main comparisons. These reference values s_{ref} are fixed benchmark constants adopted directly from the AlphaEvolve evaluation environments (Novikov et al., 2025) to ensure standardized comparisons. Note that s_{ref} represents a truncated baseline; consequently, the unclipped normalized score s_{norm} may slightly exceed 1.0 due to this precision difference.

B.1. Kissing Number (KN)

For Kissing Number, the search space consists of finite sets of nonzero integer vectors in $\mathbb{Z}^{11}$. A candidate set S is valid only if every point satisfies the norm bound

$$\|x\|_2^2 \leq 4, \qquad x \in S, \tag{7}$$

and every pair of distinct points satisfies the separation constraint

$$\|x - y\|_2^2 \geq 4, \qquad x \neq y,\ x, y \in S. \tag{8}$$

The evaluator returns the cardinality of the valid construction,

$$s_{\text{raw}} = |S|. \tag{9}$$

In our experiments, the task parameter is fixed to dimension $d = 11$, and the reference value used for normalization is $s_{\text{ref}} = 593$. The best evolved task agent shown below achieves $s_{\text{raw}} = 582$, corresponding to $s_{\text{norm}} = 582/593 = 0.9815$.

```python
# EVOLVE-BLOCK-START
import numpy as np
import itertools

def kissing_number11() -> np.ndarray:
    """
    Constructs a set of points in Z^11
    maximizing cardinality such that max
    norm <= min pairwise distance.
    Uses the M^2=4 shell strategy with a
    robust Maximum Weight Independent Set (
    MWIS) solver.
    Target: Beat 593 points.
    """
    d = 11


    # Generate all sign-complete groups
    for norm^2 <= 4.
    # Group compatibility (all points in
    G1 compatible with all points in G2)
    # is equivalent to: dot(|v1|, |v2|) <=
     (norm1^2 + norm2^2 - 4) // 2.
    gv, gs, gn = [], [], []

    # Weights 1-4 for values (+-1)
    for w in range(1, 5):
        for c in itertools.combinations(
    range(d), w):
            v = np.zeros(d, dtype=np.int32
    )
            v[list(c)] = 1
            gv.append(v)
            gs.append(2**w)
            gn.append(w)

    # Weight 1 for values (+-2)
    for i in range(d):
        v = np.zeros(d, dtype=np.int32)
        v[i] = 2
        gv.append(v)
        gs.append(2)
        gn.append(4)

    gv = np.array(gv)
    gs = np.array(gs, dtype=np.int32)
    gn = np.array(gn, dtype=np.int32)
    num_g = len(gv)

    # Precompute compatibility (adj) and
    conflict matrices
    # Condition: dot(p, q) <= (||p||^2 +
    ||q||^2 - 4) // 2
    dots = gv @ gv.T
    norms_sum = gn[:, None] + gn[None, :]
    adj = dots <= (norms_sum - 4) // 2
    conflicts = ~adj
    np.fill_diagonal(conflicts, False)

    # Metrics for heuristics
    deg = conflicts.sum(axis=1)
    wdeg = conflicts @ gs

    # MWIS solver with Migrant
    Diversification (Configuration Pool
    Seeding)
    best_sel, best_count, pool = np.zeros(
    num_g, dtype=bool), 0, []
    import time
    start_time, node_bias = time.time(),
    np.zeros(num_g, dtype=float)
```

```python
    np.fill_diagonal(adj, True)
    for seed in range(42, 100000):
        if time.time() - start_time > 280: break
        np.random.seed(seed)

        # 1. Randomized Greedy Phase with Migrant logic (Seed search from mutated good solutions)
        if pool and np.random.rand() < 0.25:
            sel = pool[np.random.randint(len(pool))].copy()
            active = np.where(sel)[0]
            if len(active) > 0: sel[np.random.choice(active, max(1, len(active)//7), replace=False)] = False
            compat = np.ones(num_g, dtype=bool)
            for idx in np.where(sel)[0]: compat &= adj[idx]
        else:
            sel, compat = np.zeros(num_g, dtype=bool), np.ones(num_g, dtype=bool)

        # Expanded Heuristics: Incorporate power-weightings for diversity
        h_choice = seed % 8
        if h_choice == 0: h = gs / (wdeg + 1.1)
        elif h_choice == 1: h = (gs**1.3) / (wdeg + 1.1)
        elif h_choice == 2: h = gs / (deg + 1.1)
        elif h_choice == 3: h = (gs**1.3) / (deg + 1.1)
        elif h_choice == 4: h = np.log2(gs + 1.0) / (deg + 1.1)
        elif h_choice == 5: h = gs / (np.sqrt(wdeg) + 1.1)
        elif h_choice == 6: h = (gs**1.5) / (wdeg + 1.1)
        else: h = np.random.rand(num_g)
        score = h * (0.7 + 0.6 * np.random.rand(num_g)) + node_bias
        order = np.argsort(-score)

        for idx in order:
            if compat[idx]:
                sel[idx] = True; compat &= adj[idx]

        # 2. Local Search Phase (1-Swap with noise-assisted sideways moves)
        for _ in range(4):
            improved = False
            for i in np.random.permutation(num_g):
                confs = np.where(conflicts[i] & sel)[0]
                w_confs = gs[confs].sum()
                if not sel[i]:
                    if gs[i] > w_confs:
                        sel[confs], sel[i] = False, True; improved = True
                    elif gs[i] == w_confs and (len(confs) > 1 or np.random.rand() < 0.1):
                        # Noise-assisted sideways swap helps escape local optima
                        sel[confs], sel[i] = False, True; improved = True
            if not improved: break

        # 3. Evaluation, Reinforcement, and Pool Management
        current_count = gs[sel].sum()
        if current_count > best_count:
            best_count, best_sel = current_count, sel.copy()
            node_bias[sel] += 0.1 # Stronger reinforcement for new global best
            if best_count >= 640: break
        if current_count >= 560: # Threshold to add configuration to the Migrant Pool
            if not any(np.array_equal(sel, p) for p in pool):
                pool.append(sel.copy())
                if len(pool) > 20: pool.pop(0) # Maintain a diverse, sliding pool
    # Reconstruct 11D points from selected sign-complete groups
    final_points = []
    for idx in np.where(best_sel)[0]:
        v = gv[idx]
        supp = np.where(v != 0)[0]
        vals = v[supp]
        for signs in itertools.product([1, -1], repeat=len(supp)):
            p = np.zeros(d, dtype=np.int64)
            p[supp] = np.array(signs) * vals
            final_points.append(p)

    return np.array(final_points)

# EVOLVE-BLOCK-END
```

Listing 1. Best evolved task program for Kissing Number (KN)

B.2. Circle Packing (CP)

For Circle Packing, the goal is to place $n = 26$ circles inside the unit square $[0, 1]^2$. A candidate solution specifies centers $c_i = (x_i, y_i)$ and radii r_i for $i = 1, \ldots, n$. Validity requires that every circle lies inside the square,

$$r_i \le x_i \le 1 - r_i, \qquad r_i \le y_i \le 1 - r_i, \tag{10}$$

and that circles do not overlap,

$$\|c_i - c_j\|_2 \ge r_i + r_j, \qquad i \ne j. \tag{11}$$

The evaluator returns the total packed radius,

$$s_{\text{raw}} = \sum_{i=1}^{26} r_i. \tag{12}$$

The reference value used for normalization is $s_{\text{ref}} = 2.6350$. The best evolved task agent shown below achieves $s_{\text{raw}} = 2.6352223118$, corresponding to $s_{\text{norm}} = 1.0001$.

```python
# EVOLVE-BLOCK-START
"""Constructor-based circle packing for n=26 circles"""
import numpy as np

def construct_packing():
    """
    Construct a specific arrangement of 26 circles in a unit square
    that attempts to maximize the sum of their radii.
    Returns:
        Tuple of (centers, radii, sum_of_radii)
        centers: np.array of shape (26, 2) with (x, y) coordinates
        radii: np.array of shape (26) with radius of each circle
        sum_of_radii: Sum of all radii
    """
    # Initialize arrays for 26 circles
    n = 26
    centers = np.zeros((n, 2))


    # Initial setup: 5-5-6-5-5 staggered grid for n=26
    rng = np.random.RandomState(42)
    centers_list = []
    rows = [5, 5, 6, 5, 5]
    for i, row_n in enumerate(rows):
        for j in range(row_n):
            centers_list.append([
                (j + 0.5) / row_n + rng.uniform(-0.002, 0.002),
                (i + 0.5) / len(rows) + rng.uniform(-0.002, 0.002)
            ])
    centers = np.clip(np.array(centers_list), 0, 1)
    # Dynamic order: Prioritize boundary circles.
    # Corner distance uses sqrt (Euclidean) for more stable weighting.
    def get_order(pts):
        d_edge = np.min(np.hstack([pts, 1 - pts]), axis=1)
        corners = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
        d_corner = np.min(np.sqrt(np.sum((pts[:, np.newaxis, :] - corners)**2, axis=2)), axis=1)
        return np.argsort(d_edge + 0.07 * d_corner)

    def calculate_radii_from(pts, start_k, current_r, current_order):
        """Vectorized greedy radius calculation with numerical stability via hypot."""
        r = current_r.copy()
        px, py = pts[:, 0], pts[:, 1]
        bounds = np.minimum(np.minimum(px, 1.0 - px), np.minimum(py, 1.0 - py))
        for k in range(start_k, n):
            idx = current_order[k]
            ri = bounds[idx]
            if k > 0:
                prev = current_order[:k]
                # Hypot is more stable for high-precision coordinate descent
                dists = np.hypot(px[idx] - px[prev], py[idx] - py[prev])
                ri = min(ri, np.min(dists - r[prev]))
            r[idx] = max(0.0, ri)
        return r

    # Initial Solution
    order = get_order(centers)
    order_lookup = {idx: i for i, idx in enumerate(order)}
    best_centers = centers.copy()
    best_radii = calculate_radii_from(best_centers, 0, np.zeros(n), order)
    best_sum = np.sum(best_radii)

    # Refinement: High-iteration Simulated Annealing (140,000 steps)
    num_steps = 140000
    for step in range(num_steps):
        progress = step / num_steps
        if step % 1200 == 0:
            order = get_order(best_centers)
            order_lookup = {idx: i for i, idx in enumerate(order)}
            best_radii = calculate_radii_from(best_centers, 0, np.zeros(n), order)
            best_sum = np.sum(best_radii)

        # Cubic cooling provides a long 'warm' tail for staggered structure refinement
        temp = 0.05 * (1.0 - progress)**3.0
        target = rng.randint(0, n)
        old_pos = best_centers[target].copy()

        # Multi-phase search: Exploration -> Refinement -> Quenching
        if progress < 0.8:
            noise = rng.normal(0, temp + 1e-10, 2) if step % 2 == 0 else rng.uniform(-temp, temp, 2)
            m_scale = 0.08
        elif progress < 0.96:
            noise = rng.normal(0, temp + 1
```

```python
e-11, 2)
            m_scale = 0.015
        else:
            noise = rng.normal(0, 5e-9, 2)
            m_scale = 0.0
        best_centers[target] = np.clip(old_pos + noise, 0, 1)

        k = order_lookup[target]
        new_radii = calculate_radii_from(best_centers, k, best_radii, order)
        new_sum = np.sum(new_radii)
        if new_sum > best_sum or (m_scale > 0 and temp > 1e-9 and rng.rand() < np.exp((new_sum - best_sum) / (temp * m_scale))):
            best_sum, best_radii = new_sum, new_radii
        else:
            best_centers[target] = old_pos

    # 3. Final Multi-Scale Micro-Quenching (High-Precision Coordinate Descent)
    # Systematic search in 16 directions (polar + cardinal) to address hexagonal packing
    angles = np.linspace(0, 2*np.pi, 16, endpoint=False)
    search_dirs = [(np.cos(a), np.sin(a)) for a in angles]
    for scale in [1e-5, 1e-7, 1e-9, 1e-11]:
        improved_scale = True
        while improved_scale:
            improved_scale = False
            for idx in range(n):
                k_start = order_lookup[idx]
                for dx, dy in search_dirs:
                    old_x, old_y = best_centers[idx]
                    best_centers[idx, 0] = np.clip(old_x + dx * scale, 0, 1)
                    best_centers[idx, 1] = np.clip(old_y + dy * scale, 0, 1)
                    nr = calculate_radii_from(best_centers, k_start, best_radii, order)
                    ns = np.sum(nr)
                    if ns > best_sum + 1e-15:
                        best_sum, best_radii, improved_scale = ns, nr, True
                    else:
                        best_centers[idx, 0], best_centers[idx, 1] = old_x, old_y
    return best_centers, best_radii, best_sum

# EVOLVE-BLOCK-END
# This part remains fixed (not evolved)

def run_packing():
    """Run the circle packing constructor for n=26"""
    centers, radii, sum_radii = construct_packing()
    return centers, radii, sum_radii


def visualize(centers, radii):
    """
    Visualize the circle packing

    Args:
        centers: np.array of shape (n, 2) with (x, y) coordinates
        radii: np.array of shape (n) with radius of each circle
    """
    import matplotlib.pyplot as plt
    from matplotlib.patches import Circle
    fig, ax = plt.subplots(figsize=(8, 8))
    # Draw unit square
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    ax.set_aspect("equal")
    ax.grid(True)
    # Draw circles
    for i, (center, radius) in enumerate(zip(centers, radii)):
        circle = Circle(center, radius, alpha=0.5)
        ax.add_patch(circle)
        ax.text(center[0], center[1], str(i), ha="center", va="center")
    plt.title(f"Circle Packing (n={len(centers)}, sum={sum(radii):.6f})")
    plt.show()

if __name__ == "__main__":
    centers, radii, sum_radii = run_packing()
    print(f"Sum of radii: {sum_radii}")
    # AlphaEvolve improved this to 2.635
    # Uncomment to visualize:
    visualize(centers, radii)
```

Listing 2. Best evolved task program for Circle Packing (CP)

B.3. Heilbronn Triangle (HT)

For Heilbronn Triangle, the task is to place $n = 11$ points inside the equilateral triangle

$$\Delta = \operatorname{conv}\left\{(0,0),\ (1,0),\ \left(\tfrac{1}{2}, \tfrac{\sqrt{3}}{2}\right)\right\}, \tag{13}$$

so as to maximize the minimum area among all triangles induced by triples of points. For a valid point set $P = \{p_1, \ldots, p_{11}\} \subset \Delta$, the evaluator computes

$$s_{\text{raw}} = \min_{1 \le i < j < k \le 11} \operatorname{Area}(p_i, p_j, p_k), \tag{14}$$

recorded in our logs as `min_area_normalized`. The reference value used for normalization in the current

experiments is $s_{\text{ref}} = 0.036529889880030156$. The best evolved task agent shown below achieves $s_{\text{raw}} = 0.0365253447400576$, corresponding to $s_{\text{norm}} = 0.9999$.

```python
# EVOLVE-BLOCK-START
import numpy as np

def heilbronn_triangle11() -> np.ndarray:
    """
    Generates an optimized arrangement of 11 points within an equilateral triangle
    to maximize the area of the smallest triangle formed by any three points.
    Vertices are at (0,0), (1,0), and (0.5, sqrt(3)/2).
    """
    n = 11
    sqrt3 = 1.7320508075688772
    h = 0.8660254037844386
    rng = np.random.RandomState(42)

    # Restore Genetic Key: High-performance vectorized search with refined targeted hill climbing
    tri_idx = np.array([[i, j, k] for i in range(n) for j in range(i + 1, n) for k in range(j + 1, n)])
    i0, i1, i2 = tri_idx.T
    pop_size, steps, rows = 64, 57000, np.arange(64)

    # Archetype 4-3-2-2 row configuration
    base = np.array([[0,0],[.33,0],[.67,0],[1,0],[.15,.25],[.
    [.85,.25],[.3,.5],[.7,.5],[.43,.73],[.57,.

    pts = np.tile(base, (pop_size, 1, 1))
    pts[1:] += rng.uniform(-0.04, 0.04, pts[1:].shape)

    # Inline initial projection and area logic
    pts[:, :, 1] = np.clip(pts[:, :, 1], 0, h)
    pts[:, :, 0] = np.clip(pts[:, :, 0], pts[:, :, 1]/sqrt3, 1.0 - pts[:, :, 1]/sqrt3)

    def calc_areas_pop(p):
        p1, p2, p3 = p[:, i0], p[:, i1], p[:, i2]
        return 0.5 * np.abs((p2[...,0]-p1[...,0])*(p3[...,1]-p1[...,1]) - (p3[...,0]-p1[...,0])*(p2[...,1]-p1[...,1]))

    areas = calc_areas_pop(pts)
    min_areas = np.min(areas, axis=1)

    for step in range(steps):
        amp = 0.12 * (0.99987 ** step)
        selector, pts_to_mut = rng.rand(pop_size), rng.randint(0, n, pop_size)

        # Targeted Mutation: 95% focus on points within the 5 smallest bottleneck triangles
        mask = selector < 0.95
        if np.any(mask):
            num_t = np.sum(mask)
            worst = np.argpartition(areas[mask], 5, axis=1)[:, :5]
            sel_tri = worst[np.arange(num_t), rng.randint(0, 5, size=num_t)]
            pts_to_mut[mask] = tri_idx[sel_tri, rng.randint(0, 3, size=num_t)]
        old_coords = pts[rows, pts_to_mut].copy()
        mut_noise = rng.standard_normal((pop_size, 2)) if step % 2 else rng.uniform(-1, 1, (pop_size, 2))
        pts[rows, pts_to_mut] += mut_noise * amp

        # Inline boundary projection
        py = np.clip(pts[rows, pts_to_mut, 1], 0, h)
        pts[rows, pts_to_mut, 1], pts[rows, pts_to_mut, 0] = py, np.clip(pts[rows, pts_to_mut, 0], py/sqrt3, 1.0 - py/sqrt3)
        new_areas = calc_areas_pop(pts)
        new_min = np.min(new_areas, axis=1)

        # Accept improvements or stability
        keep = new_min >= min_areas
        pts[rows[~keep], pts_to_mut[~keep]] = old_coords[~keep]
        areas[keep], min_areas[keep] = new_areas[keep], new_min[keep]

        # Population Sync: Clones the best configurations to the worst ones to accelerate convergence
        if step > 0 and step % 1000 == 0:
            order = np.argsort(min_areas)
            pts[order[:15]] = pts[order[-15:]].copy() + rng.uniform(-amp*0.15, amp*0.15, (15, n, 2))
            py_s = np.clip(pts[order[:15], :, 1], 0, h)
            pts[order[:15], :, 1], pts[order[:15], :, 0] = py_s, np.clip(pts[order[:15], :, 0], py_s/sqrt3, 1.0 - py_s/sqrt3)
            areas[order[:15]] = calc_areas_pop(pts[order[:15]])
            min_areas[order[:15]] = np.min(areas[order[:15]], axis=1)
    return pts[np.argmax(min_areas)]
# EVOLVE-BLOCK-END
```

Listing 3. Best evolved task program for Heilbronn Triangle (HT)