Evolutionary Multi-Task Optimization for LLM-Guided Program Discovery

Halil Alperen Gozeten [1] **Xuechen Zhang** [1] **M. Emrullah Ildiz** [1] **Ege Onur Taga** [1] **Tara Javidi** [2] **Samet Oymak** [1]

Abstract

Recent LLM-guided evolutionary search methods have shown that iterative program mutation can discover strong algorithms, but they typically optimize each task independently, even when related tasks share reusable structure. We introduce Evolutionary Multi-Task Optimization (EMO) for LLM-guided program discovery, and propose EMO-STA (Shared-Then-Adapt), a two-stage framework that first evolves a shared archive of executable programs across a task family and then adapts selected shared candidates to each target task. Within EMO-STA, we explore multiple adaptation strategies, including warm-starting from the shared archive, adapting the best average shared program, and adapting the shared program that performs best on each target task. Across eight task families spanning continuous optimization, geometric construction, modeling, and algorithmic optimization, EMO-STA improves over matched-compute single-task evolution in most settings, with STA Best-Local providing the strongest in-distribution adaptation and STA Best-Shared yielding robust transfer to unseen tasks. Compute-allocation experiments show that allocating a substantial fraction of the family-level budget to shared evolution is consistently beneficial, with roughly balanced shared and adaptation budgets often being optimal. Beyond compute efficiency, we show that shared evolution can mitigate overfitting in low-evidence settings (e.g. few training data), including ARC tasks and time-series feature engineering, by favoring programs that generalize across all tasks rather than exploiting task-specific brittle artifacts.

[1]University of Michigan, Ann Arbor [2]University of California San Diego. Correspondence to: Halil Alperen Gozeten <alperen@umich.edu>.

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).

1. Introduction

Recent LLM-guided discovery methods such as FunSearch and AlphaEvolve have shown that evolutionary code search can discover new mathematical constructions and improve scientific or algorithmic procedures. Besides applications in scientific discovery, evolutionary search (ES) has major impact in a variety of domains including agentic AI systems. For instance, ES can be used for iteratively generating code until test cases pass, prompt optimization to maximize language model performance or, more generally, optimizing the prompts and information flow/graph of agentic systems. However, the existing methods typically run evolution independently, focused on one target problem at a time, even when there are multiple tasks with structural similarities (Romera-Paredes et al., 2024; Novikov et al., 2025). Importantly, evolutionary search with LLMs is an expensive process often requiring tens of rounds to achieve satisfactory results, which highlights a need for more efficient procedures (Romera-Paredes et al., 2024; Novikov et al., 2025; Fernando et al., 2023; Agrawal et al., 2025). Although evolutionary multitasking provides a natural precedent for jointly optimizing related tasks, much of this literature focuses on shared-population search in a fixed representation space, such as unified decision-vector encodings or genetic-programming representations (Gupta et al., 2016; Scott & De Jong, 2017; Cai et al., 2021).

This leaves open the problem of open-ended text-based multitasking, specifically: how to conduct evolutionary search over LLM-generated executable programs across related tasks, while still allowing task-specific adaptation under each task's objective?

In this paper, we tackle the multitask optimization challenge with LLMs by introducing a set of algorithms under the *Shared-Then-Adapt (STA)* umbrella. EMO-STA first runs a single shared evolutionary search over the full task family, where each candidate is evaluated by a shared family-level objective that aggregates performance across tasks and produces an archive of candidate programs. Then, for each target task, EMO-STA starts a task-specific adaptation run from the shared archive using one of three initializations as detailed in Figure 1:

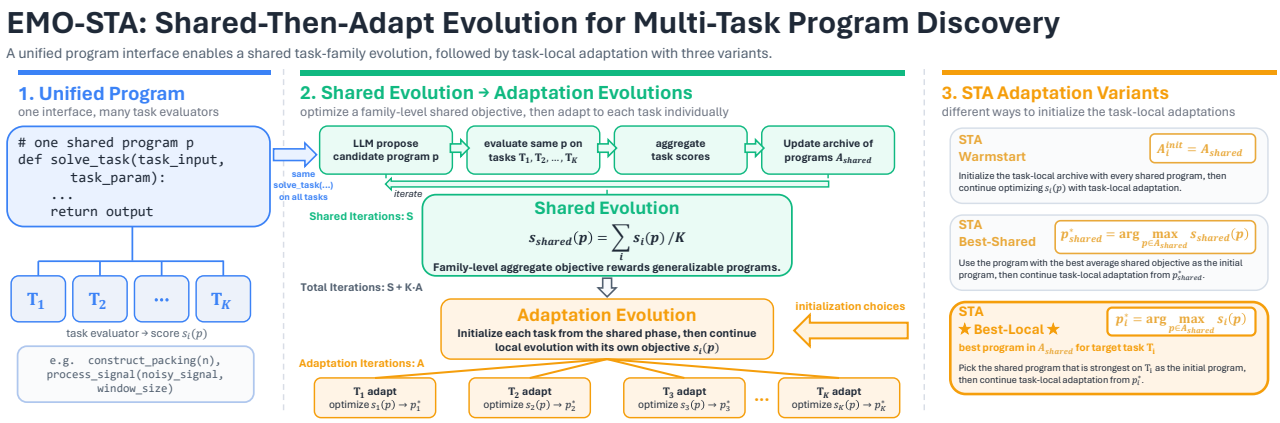


Figure 1. **EMO-STA shared-then-adapt framework.** A unified candidate-program interface lets the same evolving program p be evaluated across a related task family $\mathcal{T}_1, \ldots, \mathcal{T}_K$. EMO-STA first runs shared evolution with aggregate objective $s_{\text{shared}}(p) = \frac{1}{K}\sum_{i=1}^{K} s_i(p)$, producing a shared archive $\mathcal{A}_{\text{shared}}$ of reusable programs. It then initializes task-specific adaptation from this archive using one of three STA variants: Warmstart, Best-Shared, or **Best-Local**. The **Best-Local** variant selects, for each target task, the shared program with the highest target-task score before adaptation, and serves as our primary strategy for in-distribution task-local adaptation.

(i) *STA Warmstart*: transfers the entire shared evolution program archive into the task-local archive and continues evolution from that population;

(ii) *STA Best-Shared*: uses the program with the best average shared score across the family as the initial program for task-local evolution; and

(iii) *STA Best-Local*: selects the program in the shared archive that performs best on the target task and uses it as the initial program for adaptation.

Building on this formalism, our work makes three contributions toward more efficient and generalizable LLM-guided evolutionary optimization: **First**, we identify multitask program discovery as a key challenge for evolutionary optimization with language models and introduce EMO-STA, a shared-then-adapt framework that evolves a reusable program archive across a task family before adapting selected candidates to individual tasks. By design, EMO-STA biases the search toward a general-purpose solution as it requires the LLM to write a single program that applies to all tasks simply by varying task-specific parameters. **Second**, across diverse continuous-optimization, geometric, modeling, and algorithmic task families, we show that EMO-STA improves solution quality over matched-compute single-task evolution in most settings (see Section 3). Our ablations show that STA Best-Local is the most reliable in-distribution adaptation strategy, while allocating a substantial—and often roughly balanced—fraction of compute to shared evolution yields strong performance. **Finally**, we show that shared evolution can improve generalization: *STA Best-Shared* provides robust transfer to held-out tasks (Section 3.4), and in low-evidence settings such as ARC and time-series forecasting, shared evolution helps mitigate overfitting by favoring programs that succeed across related tasks rather than exploiting spurious task-specific artifacts (specifically, overfitting to the training dataset due to insufficient sample size, see Section 4). These out-of-distribution results suggest that, in settings where single-task evolution continues to reinforce spurious task-specific solutions, EMO-STA can offer benefits that are not merely due to additional search budget but to the structural bias (e.g., a better search space) introduced by shared evolution.

Overall, EMO-STA offers an effective multi-task evolutionary search method that converts shared structure across related problems into stronger and more generalizable LLM-discovered programs.

2. EMO-STA: Multitask Evolution with *Shared-Then-Adapt*

EMO-STA is applicable whenever related tasks can be expressed through a shared executable interface, so the same LLM-generated candidate program can be evaluated by task-specific evaluators during shared evolution and then adapted to individual tasks. We enforce this compatibility by defining a unified entry-point function and prompting the LLM to implement that format. For example, circle-packing tasks with $n = 20, 22, 24, 26$ can share `construct_packing(n)`, even though AlphaEvolve studies a fixed $n = 26$. Similarly, signal-processing tasks can vary the data generator and scoring regime across trend+sine, multi-frequency, chirp, and step-change signals while keeping the same interface `process_signal(noisy_signal, window_size)`.

Table 1. Comparison of standard single-task and EMO-STA optimization for continuous optimization families. Each score cell reports *mean* ± *std*; the first line is *STA Best-Local / STA Warmstart*, and the second line is *STA Best-Shared / Single-task*. Bold marks the largest mean among the four scores in each cell.

Model	Function minimization STA Best-Local / STA Warmstart STA Best-Shared / Single-task	Circle packing STA Best-Local / STA Warmstart STA Best-Shared / Single-task	Circle packing rectangles STA Best-Local / STA Warmstart STA Best-Shared / Single-task	Heilbronn triangle STA Best-Local / STA Warmstart STA Best-Shared / Single-task
Haiku-4.5	**.952 ± .04** / .949 ± .05 .941 ± .06 / .888 ± .05	.934 ± .03 / .926 ± .03 **.940 ± .02** / .865 ± .03	.861 ± .02 / **.865 ± .02** .845 ± .01 / .832 ± .01	**.650 ± .06** / .628 ± .05 .628 ± .06 / .547 ± .03
Sonnet-4.5	**.925 ± .02** / .917 ± .02 .904 ± .03 / .891 ± .05	**.965 ± .02** / .964 ± .02 .947 ± .03 / .927 ± .02	**.898 ± .03** / .890 ± .03 .892 ± .04 / .840 ± .02	**.622 ± .04** / .596 ± .05 .619 ± .05 / .548 ± .04
Opus-4.5	**.969 ± .03** / .942 ± .07 .941 ± .09 / .914 ± .05	**.940 ± .01** / .926 ± .01 .930 ± .01 / .912 ± .01	**.951 ± .01** / .943 ± .01 .943 ± .01 / .912 ± .01	**.741 ± .03** / .732 ± .04 .704 ± .04 / .622 ± .06
Sonnet-4.6	.988 ± .02 / .973 ± .03 **.991 ± .02** / .901 ± .02	**.997 ± .00** / **.997 ± .00** **.997 ± .00** / .957 ± .03	**.986 ± .01** / .985 ± .01 .985 ± .01 / .967 ± .02	.862 ± .04 / .809 ± .04 **.865 ± .07** / .678 ± .05
Opus-4.6	**.945 ± .03** / .943 ± .03 .932 ± .04 / .895 ± .04	**.984 ± .01** / .972 ± .02 .979 ± .02 / .963 ± .01	**.967 ± .01** / .957 ± .01 .962 ± .01 / .944 ± .01	.863 ± .03 / .844 ± .03 **.877 ± .03** / .744 ± .04

2.1. Problem Setup

We study whether a fixed family-level compute budget can be used more efficiently across related discovery tasks by first discovering reusable program structure jointly, then adapting it to each task, rather than solving each task independently. Let $\mathfrak{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_K\}$ be a family of related discovery tasks. In EMO-STA, we assume that all tasks in a family are written so that the same candidate program can be executed on each of them. Equivalently, the family shares one program space $\mathcal{P}$ and one executable interface, while each task $\mathcal{T}_i$ has its own evaluator and score $s_i(\cdot)$. When raw objectives differ in scale across tasks, we define s_i as a task-normalized score, so that the shared objective averages scores on a common scale rather than being biased toward tasks with larger numerical ranges. Given a total family-level iteration budget B_{tot}, our goal in shared evolution is to produce task-specific solutions $p_1, \ldots, p_K \in \mathcal{P}$ that maximize average family performance,

$$\max_{p_1,\ldots,p_K \in \mathcal{P}} \quad \frac{1}{K}\sum_{i=1}^{K} s_i(p_i)$$
$$\text{subject to} \quad \text{total iteration budget} \le B_{\text{tot}}.$$

where $s_i(\cdot)$ is the possibly normalized score on task $\mathcal{T}_i$. Equivalently, we can view the problem as one of *sample efficiency*: can we reach a target quality across the family using fewer evaluations than solving each task independently?

2.2. Shared Evolution Phase

In the shared phase, we evolve a single archive against the aggregated family-level objective:

$$s_{\text{shared}}(p) = \frac{1}{K}\sum_{i=1}^{K} s_i(p).$$

Under this objective, the best shared program is the candidate with the highest average performance across the task family. Thus, the shared search favors program structures that capture reusable cross-task structure, rather than structures tailored to a single task. Beyond the single best shared program, the shared phase produces a shared archive of programs, denoted by $\mathcal{A}^{\text{shared}}$. The goal of this phase is not to force all tasks to share one final solution, but to obtain a set of candidate programs that can later be converted into task-local initializations for adaptation.

2.3. Adaptation Evolution Phase

Starting from the shared archive $\mathcal{A}^{\text{shared}}$, EMO-STA constructs task-local initialization checkpoints for each target task. For each target task $\mathcal{T}_i$, we convert shared candidates into task-local checkpoint entries through a rescoring projection, denoted by $\Pi_i : \mathcal{A}^{\text{shared}} \to \mathcal{A}_i^{\text{init}}$. For a shared program p, $\Pi_i(p)$ is the same program paired with its task-local evaluation state, obtained by re-evaluating the code under the evaluator for $\mathcal{T}_i$. Using this projection, we consider three ways to initialize the task-local adaptation evolution.

STA Warmstart. This method transfers the entire shared program archive into the task-local archive and continues adaptation evolution from the resulting population:

$$\mathcal{A}_{i,\text{warm}}^{\text{init}} = \{\Pi_i(p) : p \in \mathcal{A}^{\text{shared}}\}.$$

STA Best-Shared. This method uses the program with the best shared family-level score as the initial program for adaptation evolution:

$$p_{\text{bs}}^{\star} = \arg\max_{p \in \mathcal{A}^{\text{shared}}} s_{\text{shared}}(p), \qquad \mathcal{A}_{i,\text{bs}}^{\text{init}} = \{\Pi_i(p_{\text{bs}}^{\star})\}.$$

STA Best-Local. This method uses the program in the shared archive that performs best on the target task $\mathcal{T}_i$ as the

Table 2. Comparison of standard single-task and EMO-STA optimization for modeling and algorithmic optimization families. Each score cell reports *mean* ± *std*; the first line is *STA Best-Local / STA Warmstart*, and the second line is *STA Best-Shared / Single-task*. Bold marks the largest mean among the four scores in each cell.

Model	Signal processing STA Best-Local / STA Warmstart STA Best-Shared / Single-task	SLDBench-3D STA Best-Local / STA Warmstart STA Best-Shared / Single-task	Rust adaptive sort STA Best-Local / STA Warmstart STA Best-Shared / Single-task	K-module STA Best-Local / STA Warmstart STA Best-Shared / Single-task
Haiku-4.5	$\mathbf{.600 \pm .05}$ / $.584 \pm .06$ $.597 \pm .04$ / $.569 \pm .01$	$\mathbf{.958 \pm .02}$ / $.953 \pm .02$ $.949 \pm .02$ / $.951 \pm .01$	$.533 \pm .02$ / $.535 \pm .02$ $.509 \pm .03$ / $\mathbf{.539 \pm .02}$	$.567 \pm .06$ / $.567 \pm .04$ $\mathbf{.575 \pm .07}$ / $.550 \pm .03$
Sonnet-4.5	$\mathbf{.587 \pm .01}$ / $.578 \pm .02$ $.582 \pm .02$ / $.576 \pm .01$	$\mathbf{.976 \pm .01}$ / $.971 \pm .01$ $.971 \pm .02$ / $.959 \pm .01$	$.481 \pm .03$ / $.484 \pm .03$ $.457 \pm .03$ / $\mathbf{.528 \pm .01}$	$.617 \pm .03$ / $\mathbf{.650 \pm .02}$ $.567 \pm .06$ / $.617 \pm .05$
Opus-4.5	$.620 \pm .03$ / $\mathbf{.635 \pm .03}$ $.625 \pm .02$ / $.568 \pm .01$	$\mathbf{.983 \pm .00}$ / $.972 \pm .01$ $.981 \pm .00$ / $.973 \pm .01$	$.515 \pm .05$ / $\mathbf{.520 \pm .05}$ $.483 \pm .05$ / $.497 \pm .02$	$.617 \pm .03$ / $\mathbf{.675 \pm .03}$ $.592 \pm .03$ / $.567 \pm .05$
Sonnet-4.6	$\mathbf{.628 \pm .04}$ / $.626 \pm .04$ $.613 \pm .05$ / $.608 \pm .03$	$.969 \pm .01$ / $.968 \pm .01$ $\mathbf{.969 \pm .01}$ / $.955 \pm .01$	$.659 \pm .01$ / $\mathbf{.663 \pm .01}$ $.656 \pm .01$ / $.616 \pm .03$	$.617 \pm .09$ / $\mathbf{.700 \pm .07}$ $.575 \pm .03$ / $.675 \pm .05$
Opus-4.6	$.713 \pm .05$ / $.707 \pm .04$ $\mathbf{.716 \pm .04}$ / $.648 \pm .03$	$\mathbf{.975 \pm .01}$ / $.973 \pm .01$ $.967 \pm .01$ / $.964 \pm .02$	$.616 \pm .02$ / $\mathbf{.625 \pm .02}$ $.612 \pm .02$ / $.531 \pm .05$	$.725 \pm .02$ / $\mathbf{.800 \pm .05}$ $.692 \pm .05$ / $.758 \pm .08$

initial program for task-local evolution:

$$p^{\star}_{i,\mathrm{bl}} = \arg\max_{p \in \mathcal{A}^{\mathrm{shared}}} s_i(p), \qquad \mathcal{A}^{\mathrm{init}}_{i,\mathrm{bl}} = \{\Pi_i(p^{\star}_{i,\mathrm{bl}})\}.$$

Together, these variants represent different ways of transferring the shared search state into task-local adaptation. *STA Warmstart* preserves archive diversity, *STA Best-Shared* starts from the strongest average family-level program, and *STA Best-Local* starts from the shared candidate that already performs best on the target task.

Each initialized archive $\mathcal{A}^{\mathrm{init}}_{i,v}$, $v \in \{\mathrm{warm}, \mathrm{bs}, \mathrm{bl}\}$, is then evolved independently on task $\mathcal{T}_i$ with the same adaptation budget. Thus, the three EMO-STA variants share the same shared phase, evaluator family, and budget; only the task-local initialization differs. Algorithm 1 in Appendix A summarizes the full shared-then-adapt procedure.

Reported scores and compute matching. For each task $\mathcal{T}_i$, we report four scores: the three adapted EMO-STA variants initialized from $\mathcal{A}^{\mathrm{init}}_{i,\mathrm{warm}}$, $\mathcal{A}^{\mathrm{init}}_{i,\mathrm{bs}}$, and $\mathcal{A}^{\mathrm{init}}_{i,\mathrm{bl}}$, respectively, and the *Single-task* baseline, obtained by evolving $\mathcal{T}_i$ from scratch without access to a shared archive.

To compare methods under a fair family-level total compute, let S denote the shared-phase iterations, A the per-task adaptation iterations, and K the number of tasks. We choose the per-task single-task iterations B so that $S + KA = KB$. Thus, shared-then-adapt search and K independent single-task runs use the same total evolutionary budget, so the comparison isolates whether shared program structure improves task scores compared to solving tasks independently.

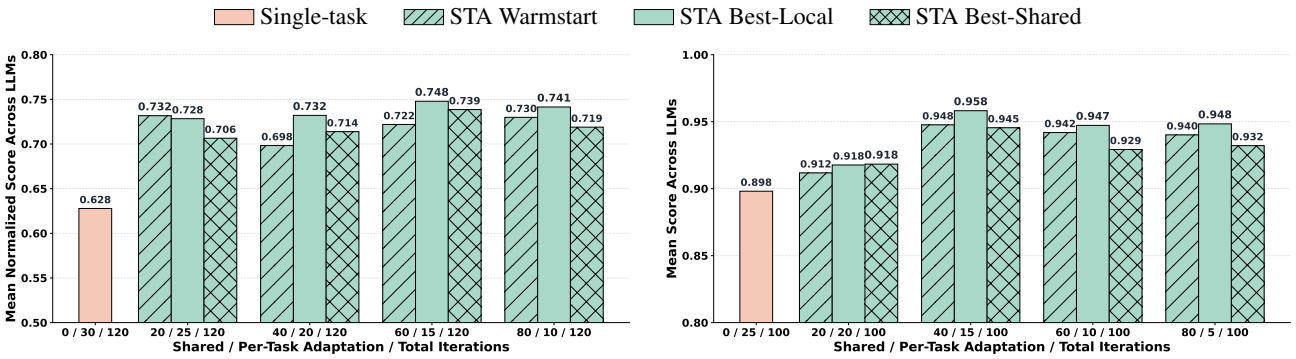
3. Empirical Evaluation of EMO-STA

We evaluate EMO-STA on eight task families, each defined by related subtasks, spanning continuous optimization, geometric construction, modeling, and algorithmic tasks: function minimization, circle packing, circle-packing rectangles, Heilbronn triangle, signal processing, SLDBench-3D, Rust adaptive sort, and K-module. Our experiments address four questions: (i) whether STA improves over single-task optimization under matched compute, (ii) how the gains distribute across tasks within a family, (iii) how a fixed family budget should be divided between shared and adaptation evolutions, and (iv) whether the evolved programs generalize to held-out task sizes in OOD evaluation. Across main experiments, we report results for five models: Claude Haiku-4.5, Sonnet-4.5, Sonnet-4.6, Opus-4.5, and Opus-4.6. For family-level results, we first average scores over in-distribution tasks in each family for each run, then report mean ± std over five independent runs, with all reported scores higher-is-better. Task-level figures report subtask scores separately.

3.1. Experimental setup

We first describe the implementation setup, task-family construction, score reporting, and compute matching used across all experiments.

Implementation and task interface unification. All experiments are implemented on top of OpenEvolve, an open-source framework for AlphaEvolve-style evolutionary code search (Sharma, 2025; Novikov et al., 2025). We keep its underlying program-evolution loop, including island-based population management, MAP-Elites context selection, and checkpointing, and implement EMO-STA on top of that infrastructure. We apply EMO-STA to task families adaptable to a common program interface, so the same evolving program can be evaluated across all subtasks in the family. For example, in circle packing



(a) Heilbronn triangle family with 120 total iterations. The best setting is *STA Best-Local* at $60/15/120$.

(b) Function minimization family with 100 total iterations. The best setting is *STA Best-Local* at $40/15/100$.

Figure 2. Compute-allocation results for EMO-STA on two $K = 4$ task families. Grouped bars compare *STA Warmstart*, *STA Best-Local*, and *STA Best-Shared* across *Shared / Per-task Adapt / Total* allocations. The leftmost bar is the direct single-task baseline with allocation $0/B/KB$. Bars report mean scores averaged over Claude Haiku-4.5, Sonnet-4.5, Sonnet-4.6, Opus-4.5, and Opus-4.6.

the same evolving program is used across all task sizes through `construct_packing(n)`, with the evaluator varying only the current task parameter n. Representative prompts specifying these shared interfaces are provided in Appendix D.1.

Task families, baselines, and compute matching. Most task families contain four in-distribution training tasks, while SLDBench-3D contains two. Within each family, the shared phase, evaluator family, and adaptation budget are identical across *STA Warmstart*, *STA Best-Shared*, and *STA Best-Local*; only the task-local adaptation initialization differs. Thus, these variants ablate how the shared archive is used to initialize adaptation. The *Single-task* baseline evolves each task independently from scratch, using the same program interface but without shared evolution or shared-archive access. For fair comparison, we use matched family-level iteration budgets: if the shared phase uses S iterations, each task-specific adaptation uses A, and a family contains K tasks, then the per-task single-task budget B satisfies $S + KA = KB$. We report budgets as *Shared / Per-task Adapt / Total*, where *Total* denotes $S + KA = KB$. In the reported table settings, the total budget is 120 iterations for circle packing, circle packing rectangles, Heilbronn triangle, and K-module; 100 for function minimization, signal processing, and Rust adaptive sort; and 80 for SLDBench-3D, which contains $K = 2$ tasks. These totals are the matched-compute settings used in the main analysis, but similar qualitative trends hold across different total iteration budgets, as shown in Appendix B.4.

Score normalization and evaluation design. For the three geometric families—circle packing, circle-packing rectangles, and Heilbronn triangle—different task sizes have different objective scales. To make shared evolution meaningful, we normalize each task score by its corresponding known target value, so that the shared objective averages comparable normalized scores rather than being dominated by larger-n tasks. As concrete examples, for circle-packing families we divide by fixed per-n reference sums of radii, and for Heilbronn triangle we divide by fixed per-n reference minimum triangle areas. Under this convention, a score of 1 corresponds to matching the reference value for that task size, while values below or above 1 indicate performance below or above that reference. These normalized scores are also the values reported in the tables for those three families.

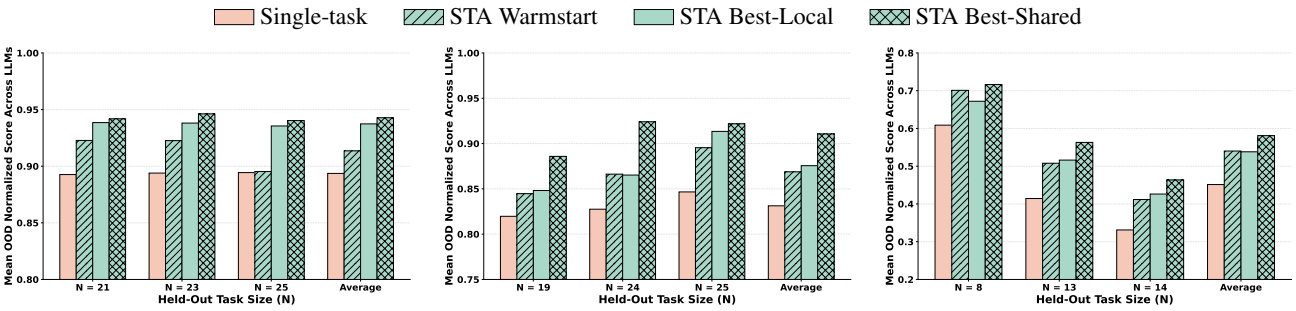
Additional details on task-family construction, score normalization, held-out task sizes, and compute-allocation settings are provided in Appendix C.1.

3.2. In-Distribution Results and Task-Level Transfer

We begin with the in-distribution comparison against direct single-task evolution, then inspect how improvements vary across individual subtasks.

EMO-STA consistently improves over single-task optimization. Tables 1 and 2 compare EMO-STA against single-task evolution under matched family-level compute. Across both tables, at least one EMO-STA variant exceeds the single-task baseline in $38/40$ cells. The improvements are especially large on geometric tasks such as Heilbronn triangle, where performance improves from 0.547 to 0.650 for Haiku-4.5, from 0.622 to 0.741 for Opus-4.5, and from 0.744 to 0.877 for Opus-4.6. EMO-STA also improves when the single-task baseline is already competitive, e.g., from 0.957 to 0.997 on circle packing with Sonnet-4.6, from 0.944 to 0.967 on circle-packing rectangles with Opus-4.6, and from 0.964 to 0.975 on SLDBench-3D with Opus-4.6.

***STA Best-Local* is the most reliable adaptation strategy.** Across in-distribution results, *STA Best-Local* achieves the highest average score, is strongest in $23/40$ cells, and outperforms the *Single-task* baseline in $35/40$ cells. This is intuitive because the shared archive contains multiple useful



(a) Circle packing. *(b) Circle packing rectangle.* *(c) Heilbronn triangle.*

Figure 3. Held-out task-size evaluation at fixed *60 / 15 / 120* compute allocation across three domains: *circle packing*, *circle packing rectangle*, and *Heilbronn triangle*. For each STA variant, programs are adapted to each in-distribution task size and evaluated on each held-out size; bars report mean OOD score across adaptation source tasks and models. The comparison includes the *Single-task* baseline and the STA variants: *STA Warmstart*, *STA Best-Local*, and *STA Best-Shared.*

candidates, and the best starting point for a task is often not the program with the highest average family-level score, but the one that already performs best on that task. By average score, *STA Warmstart* is next, followed by *STA Best-Shared*; both remain competitive, with *STA Best-Shared* outperforming the *Single-task* baseline in $33/40$ cells.

Some task structures favor archive-level transfer. Not all families prefer the same transfer mode, and in K-module, archive-level transfer with *STA Warmstart* is often strongest. This is consistent with K-module being a discrete trial-and-error task, where preserving multiple shared-archive candidates can be more useful than starting adaptation from a single best task-local candidate. In Rust adaptive sort, weaker models struggle to translate shared evolution into task-specific gains, while stronger models benefit from the shared archive with *STA Warmstart* and improve over the single-task baseline.

Task-level gains confirm the family-level improvement trends. Figure 4a breaks down the *STA Best-Local* results by individual tasks, showing how gains vary across them. The open markers show the pre-adaptation best shared program score and the filled markers show the final adapted score, both measured relative to the single-task baseline. Across all subtasks, the filled markers move to the right of the open markers and lie above zero, showing that task-local adaptation consistently improves the shared candidate and turns family-level structure into task-level gains. This indicates that EMO-STA is not merely improving family-level averages through one outlier task; rather, shared search followed by task-local adaptation improves many subtasks within each family. Also, Appendix D.2 gives concrete program examples of this behavior, where shared evolution finds a reusable solver structure and *STA Best-Local* refines it for a target task.

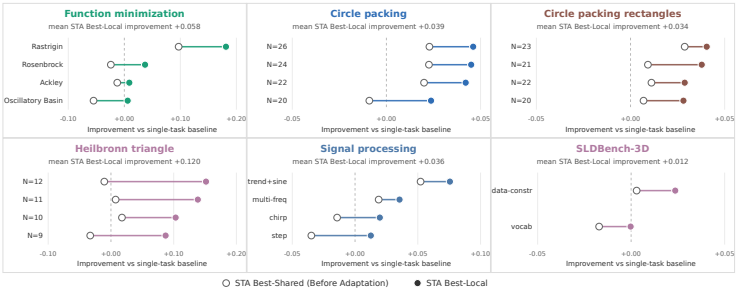
3.3. Compute Allocation: Shared vs. Adaptation Evolutions

Figure 2 compares how a family-level compute budget is divided between shared evolution and per-task adaptation. Across the comparisons in Figure 2 and Appendix B.4, shared evolution is consistently beneficial, with nonzero shared-budget configurations outperforming the single-task baseline for all three EMO-STA variants.

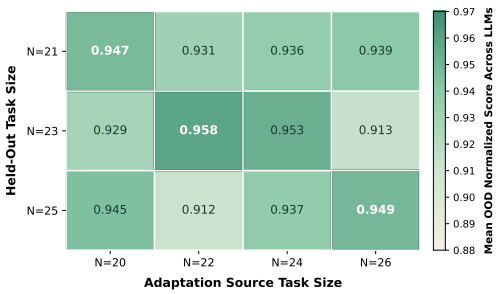
Equal shared/adaptation compute splits are consistently strong. A particularly effective allocation recipe is to split compute roughly evenly between shared evolution and the aggregate task-local adaptation budget, i.e., $S \approx KA$. In Figure 2a, the equal-split setting $60/15/120$ is the best setting for Heilbronn triangle, while Figure 2b shows that function minimization peaks nearby at $40/15/100$. We note that the equal-split point or a nearby allocation provides a strong shared/adaptation trade-off across all *STA Warmstart*, *STA Best-Shared*, and *STA Best-Local*. Additional compute-allocation results in Appendix B.4 show the same qualitative pattern on the two circle-packing families. Intuitively, this balance gives the shared phase enough budget to discover reusable family-level structure, while preserving sufficient task-local compute to specialize that structure to each task.

3.4. Generalization to Held-Out Task Sizes

We evaluate out-of-distribution (OOD) transfer in geometric families. For each EMO-STA variant, we take programs adapted to each in-distribution task, evaluate them on each held-out task, and in Figure 3 report the mean across adaptation source tasks. The in-distribution / held-out sizes are $N = \{20, 22, 24, 26\}/\{21, 23, 25\}$ for circle packing, $N = \{20, 21, 22, 23\}/\{19, 24, 25\}$ for circle packing in rectangles, and $N = \{9, 10, 11, 12\}/\{8, 13, 14\}$ for Heilbronn triangle. Additional OOD results in Appendix B.3 break down the held-out evaluations across shared/adaptation compute allocations, showing similar overall trends.



(a) Task-level transfer gains for *STA Best-Local*. Each panel shows one task family and each row one in-distribution task. Open markers show the pre-adaptation shared score, filled markers show the final adapted score, and the x-axis reports improvement over the single-task baseline.



(b) OOD circle-packing transfer for *STA Best-Local* at $60/15/120$. Rows are held-out sizes, columns are source tasks, and cells report mean OOD normalized score across LLMs and seeds.

Figure 4. Task-level and held-out transfer behavior of *STA Best-Local*. Left: in-distribution adaptation gains across task families. Right: circle-packing OOD transfer from each in-distribution adaptation source task to each held-out task size.

***STA Best-Shared* gives the strongest held-out transfer.** Figure 3 evaluates in-distribution programs on held-out task sizes for the three geometric families. Across circle packing, circle packing in rectangles, and Heilbronn triangle, *STA Best-Shared* is the strongest OOD variant on average. This suggests that the best shared evolution program often captures the most transferable family-level structure, even when *STA Best-Local* is stronger for in-distribution adaptation. The gap is especially clear for circle packing in rectangles and Heilbronn triangle, where *STA Best-Shared* outperforms both *STA Warmstart* and *STA Best-Local* across held-out sizes.

Nearby task sizes transfer best. The circle-packing heatmap in Figure 4b shows how *STA Best-Local* transfer depends on the source task used for adaptation. Nearby task sizes tend to transfer best as adaptation on $N = 20$ gives the strongest result for held-out $N = 21$ (0.947), adaptation on $N = 22$ is best for held-out $N = 23$ (0.958), and adaptation on $N = 26$ is best for held-out $N = 25$ (0.949). This illustrates that EMO-STA learns reusable family-level programs, while task-local adaptation still tunes the solution toward a particular region of the task family.

4. Mitigating Overfitting with Shared Evolution

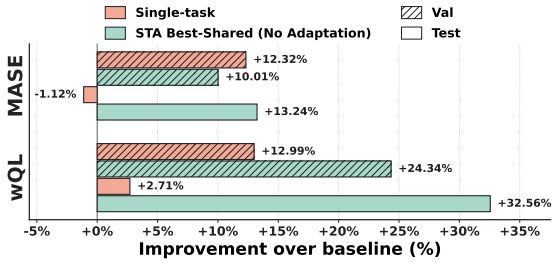
In this section, we examine settings where the benefit of shared evolution goes beyond compute efficiency. When each individual task provides limited or noisy evidence, independent evolution can overfit the task-specific objective itself, so additional single-task evolution may reinforce spurious solutions rather than improve generalization. We investigate the benefit of shared evolution across two different settings: Abstraction and Reasoning Corpus (ARC) (Chollet, 2019; Xu et al., 2023) and Time-series feature engineering. In ARC, this appears as programs that fit the few training examples but fail the hidden test grid; in time-series forecasting, it appears as validation improvements that do not transfer to held-out test windows. Across both settings, shared evolution acts as a regularizer by favoring programs or transformations that work across related tasks rather than only one sparse target.

4.1. ARC

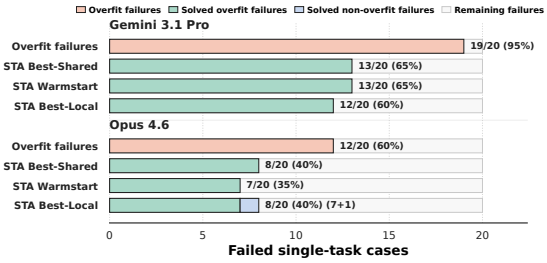
We evaluate EMO-STA on the Abstraction and Reasoning Corpus (ARC), a natural setting for studying overfitting in program evolution. Because each ARC task has only a few input–output training pairs, single-task search can find programs that fit the examples without capturing the underlying rule. We use the ARC Prize 2025 ARC-AGI evaluation split. To construct a controlled multi-task setting, we generate transformed variants of each ARC task using invertible spatial and symbolic transformations, such as rotations, flips, and color permutations. These transformations change the surface appearance of the grids while preserving the same reasoning rule up to the corresponding transformation. Additional setup details are provided in Section C.2.

Overfitting in Single-Task Evolution. We first run the single-task evolution baseline on ARC tasks separately for each primary model and select the first 20 failed run–task instances in evaluation-split order for each model. As shown in Figure 5b, overfitting is the dominant failure mode, especially for Gemini-3.1-Pro-Preview: $19/20$ Gemini failures and $12/20$ Claude Opus 4.6 failures fit all training examples but fail to generalize to the held-out test input.

Multi-Task Evolution with EMO-STA Variants. Motivated by this diagnosis, we evaluate whether structured task diversity can reduce overfitting on the same selected failure cases. For each original ARC task, EMO-STA jointly evolves over the original task and its transformed variants, allowing information to be shared across tasks that follow the same underlying rule. We evaluate all three archive-initialization variants under the same shared phase and adap-



(a) COVID Deaths. Single-task evolution transfers poorly to held-out test windows, while *STA Best-Shared* applies one shared transformation and achieves stronger test gains.



(b) ARC. Across both models, EMO-STA variants recover many failed single-task cases through shared evolution over transformed task variants, especially for overfit failures.

Figure 5. Shared evolution mitigates overfitting in two low-sample settings. In time-series feature engineering, one shared transformation improves held-out test performance over per-series evolution. In ARC, transformation-based shared evolution mainly resolves training-example overfitting.

tation budget.

As shown in Figure 5b, all EMO-STA variants recover a substantial fraction of failed single-task cases. For Gemini-3.1-Pro-Preview, *STA Best-Shared* and *STA Warmstart* each solve $13/20$ cases, while *STA Best-Local* solves $12/20$. For Claude Opus 4.6, *STA Best-Shared* and *STA Best-Local* each solve $8/20$, while *STA Warmstart* solves $7/20$. Since almost all recovered cases come from overfit failures, these results suggest that transformation-based shared evolution provides useful regularization to obtain solutions that generalize beyond sparse training examples.

4.2. Time-series Feature Engineering

We next use the EFE-Time setting from concurrent anonymized work (Authors, 2026) as a data-science testbed for overfitting in evolutionary optimization. EFE-Time evolves preprocessing programs for time-series forecasting, where each candidate transformation is selected by validation forecasting performance. This setting is susceptible to overfitting under limited data: an evolution run can discover transformations that improve the validation windows of an individual series while failing to capture structure that transfers to held-out windows. We evaluate this effect on GIFT-Eval's COVID Deaths task, a daily healthcare benchmark with 266 univariate series and a 30-step prediction horizon (Aksu et al., 2024). The target is confirmed COVID-19 deaths, whose daily counts are noisy and heterogeneous because of underreporting, cross-country reporting differences, reporting delays, and day-of-week effects.

Shared vs. per-series evolution. We treat each series as a related task. The single-task baseline evolves a separate EFE-Time transformation for each series, while STA Best-Shared evolves one family-level transformation across all series and applies it directly, with no task-local adaptation. We match budgets by running each single-task evolution for 5 iterations, for $266 \times 5 = 1330$ total iterations, and running STA Best-Shared for 1330 shared iterations. Because MASE and wQL are lower-is-better error metrics, we report improvement as percentage error reduction relative to the baseline; positive values therefore indicate lower error.

Validation–test generalization. Figure 5a shows a validation–test gap. Single-task evolution improves validation MASE and wQL by 12.32% and 12.99%, but these gains do not transfer: test MASE worsens by 1.12% and test wQL improves by only 2.71%. In contrast, STA Best-Shared without adaptation improves test MASE and wQL by 13.24% and 32.56%. This suggests that per-series evolution over-specializes to idiosyncratic reporting artifacts, while shared evolution acts as a regularizer by selecting transformations that work across related series.

5. Related Work

LLM-based evolutionary optimization and program search. Evolutionary search over programs has a long history, with genetic programming and Cartesian genetic programming showing that executable or graph-structured programs can be optimized directly against a fitness function (Koza, 1992; Miller & Thomson, 2000). Recent LLM-based systems revisit this idea by using language models to propose program edits, prompts, or search operators inside evaluation-driven optimization loops. FunSearch demonstrated this paradigm for mathematical discovery by evolving short programs under an automated evaluator, producing new cap-set constructions and improved online bin-packing heuristics (Romera-Paredes et al., 2024). AlphaEvolve broadened this to an evolutionary coding agent that edits executable code across mathematical, scientific, and systems problems (Novikov et al., 2025), while OpenEvolve provides a close open-source implementation and serves as the base system for our experiments (Sharma, 2025). Recent work has further improved the evolutionary loop itself. For example, GEPA uses reflective prompt evolution, ThetaEvolve studies test-time learning for open optimization problems, and DeltaEvolve summarizes program changes

as semantic deltas (Agrawal et al., 2025; Wang et al., 2025; Jiang et al., 2026). Complementary systems adapt higher-level search behavior during the run: AdaEvolve dynamically allocates resources and adjusts exploration, while EvoX meta-evolves the search strategy itself (Cemri et al., 2026; Liu et al., 2026). These works show the strength of LLM-based evolutionary optimization, but they mainly improve individual discovery runs or search policies within runs; they do not organize related tasks into a unified multi-task workflow that exploits shared program structure, as EMO-STA does.

Evolutionary multitasking and transfer. Evolutionary multitasking studies how related optimization tasks can be solved jointly by transferring information across tasks. Multifactorial evolution formalizes this idea through a shared population and implicit genetic transfer across tasks (Gupta et al., 2016), while multitask Cartesian genetic programming shows that program-evolution methods can also benefit from solving multiple related program-synthesis tasks in a shared representation (Scott & De Jong, 2017). A central challenge in this literature is controlling transfer so that related tasks exchange useful information without inducing negative transfer. MFEA-II addresses this by estimating inter-task transfer parameters online (Bali et al., 2020), while later methods learn explicit mappings or hybrid transfer mechanisms between tasks (Feng et al., 2019; Cai et al., 2021). EMO-STA is closest in spirit to this multi-task optimization literature, but differs in the object being transferred: rather than sharing vectors, chromosomes, or fixed symbolic genotypes, we evolve full executable programs produced by LLMs and evaluated by task-specific harnesses. Our contribution is to bring multitask transfer into LLM-based evolutionary code search by organizing related discovery tasks around a shared program interface, evolving a common archive of executable programs, and using the resulting archive to initialize task-local adaptation runs.

Warm-start and transfer in black-box optimization. Warm-starting and population seeding provide a direct precedent for EMO-STA's shared-then-adapt design. Classical initialization work shows that the choice of an evolutionary algorithm's initial population can strongly affect search efficiency, and that problem-dependent seeding can improve multi-objective evolutionary optimization, but with benefits that depend on the problem class and algorithm (Kazimipour et al., 2014; Friedrich & Wagner, 2014; Nomura et al., 2020). Bayesian optimization offers an adjacent black-box optimization perspective: multi-task BO transfers observations through a multi-task Gaussian-process surrogate (Swersky et al., 2013), warm-start BO models sequences of related optimization problems jointly (Poloczek et al., 2016), and scalable hyperparameter transfer learning replaces cubic GP transfer with a multi-task adaptive Bayesian linear regression model coupled by a shared neural representation (Feurer et al., 2022); recent surveys further organize transfer BO around initial design, search-space design, surrogate modeling, and acquisition-function transfer. EMO-STA differs from these lines in the object and timing of transfer: rather than transferring numeric seeds, covariance distributions, surrogate-model observations, or hand-designed domain building blocks, it first evolves a family-level archive of executable LLM-generated programs and then initializes task-local evolutionary adaptation from that archive, allowing shared algorithmic structure to be reused while still permitting specialization to each target task.

6. Discussion

EMO-STA is a framework for LLM-guided evolutionary program discovery: it evolves a reusable archive across a related task family and uses that archive to initialize task-local adaptations. Across our experiments, EMO-STA improves over matched-compute single-task evolution in most settings, with *STA Best-Local* strongest for in-distribution adaptation and *STA Best-Shared* more robust on held-out task sizes. The ARC and time-series case studies further suggest that shared evolution can regularize low-sample optimization by favoring programs that generalize across related tasks. A limitation of EMO-STA is that the framework benefits from tasks being related so that a reusable program can address them simply by varying task parameters. It would be desirable to handle arbitrary task sets by automatically grouping them, as well as to understand whether an LLM can infer reusable program structure by capturing what *related task* means.

Acknowledgements

We thank Konstantinos D. Polyzos, Nikhil Vaidyanathan, and Xihe Gu for helpful discussions and feedback. This work is supported in part by the NSF grants CCF-2046816, CCF-2403075, and CCF-2212426; the ONR grants N00014-24-1-2289 and N00014-22-1-2363; the Eric and Wendy Schmidt AI in Science Postdoctoral Fellowship; the NSF TILOS AI Institute; and the UCSD Centers for Machine Intelligence, Computing, and Security (MICS). Computational aspects of the research are generously supported by an Amazon Research Award on Foundation Model Development.

References

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. GEPA: Reflective prompt evolution can outperform reinforcement learn-

ing, 2025. URL https://arxiv.org/abs/2507.19457.

Aksu, T., Woo, G., Liu, J., Liu, X., Liu, C., Savarese, S., Xiong, C., and Sahoo, D. Gift-eval: A benchmark for general time series forecasting model evaluation. *arxiv preprint arxiv:2410.10393*, 2024.

Akyürek, E., Damani, M., Zweiger, A., Qiu, L., Guo, H., Pari, J., Kim, Y., and Andreas, J. The surprising effectiveness of test-time training for few-shot learning. *arXiv preprint arXiv:2411.07279*, 2024.

Ansari, A. F., Shchur, O., Küken, J., Auer, A., Han, B., Mercado, P., Rangapuram, S. S., Shen, H., Stella, L., Zhang, X., et al. Chronos-2: From univariate to universal forecasting. *arXiv preprint arXiv:2510.15821*, 2025.

Authors, A. Evolutionary feature engineering for structured data, 2026. Concurrent submission. Anonymized manuscript included in supplementary material.

Bali, K. K., Ong, Y.-S., Gupta, A., and Tan, P. S. Multifactorial evolutionary algorithm with online transfer parameter estimation: Mfea-ii. *IEEE Transactions on Evolutionary Computation*, 24(1):69–83, 2020. doi: 10.1109/TEVC.2019.2906927.

Cai, Y., Peng, D., Liu, P., and Guo, J.-M. Evolutionary multi-task optimization with hybrid knowledge transfer strategy. *Information Sciences*, 580:874–896, 2021. doi: 10.1016/j.ins.2021.09.021.

Cemri, M., Agrawal, S., Gupta, A., Liu, S., Cheng, A., Mang, Q., Naren, A., Erdogan, L. E., Sen, K., Zaharia, M., et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Chollet, F. On the measure of intelligence. *arXiv preprint arXiv:1911.01547*, 2019. URL https://arxiv.org/abs/1911.01547.

Feng, L., Zhou, L., Zhong, J., Gupta, A., Ong, Y.-S., Tan, K. C., and Qin, A. K. Evolutionary multitasking via explicit autoencoding. *IEEE Transactions on Cybernetics*, 49(9):3457–3470, 2019. doi: 10.1109/TCYB.2018.2845361.

Fernando, C., Banarse, D., Michalewski, H., Osindero, S., and Rocktäschel, T. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023. URL https://arxiv.org/abs/2309.16797.

Feurer, M., Letham, B., Hutter, F., and Bakshy, E. Practical transfer learning for bayesian optimization, 2022. URL https://arxiv.org/abs/1802.02219.

Friedman, E. Circles in rectangles of perimeter 4. https://erich-friedman.github.io/packing/cirRrec/, 2026a. Accessed: 2026-04-30.

Friedman, E. Circles in squares. https://erich-friedman.github.io/packing/cirRsqu/, 2026b. Accessed: 2026-04-30.

Friedrich, T. and Wagner, M. Seeding the initial population of multi-objective evolutionary algorithms: A computational study, 2014. URL https://arxiv.org/abs/1412.0307.

Gupta, A., Ong, Y.-S., and Feng, L. Multifactorial evolution: Toward evolutionary multitasking. *Trans. Evol. Comp*, 20 (3):343–357, June 2016. ISSN 1089-778X. doi: 10.1109/TEVC.2015.2458037. URL https://doi.org/10.1109/TEVC.2015.2458037.

Jamil, M. and Yang, X.-S. A literature survey of benchmark functions for global optimization problems. *International Journal of Mathematical Modelling and Numerical Optimisation*, 4(2):150–194, 2013. doi: 10.1504/IJMMNO.2013.055204.

Jiang, J., Ding, T., and Zhu, Z. DeltaEvolve: Accelerating scientific discovery through momentum-driven evolution, 2026. URL https://arxiv.org/abs/2602.02919.

Kazimipour, B., Li, X., and Qin, A. K. A review of population initialization techniques for evolutionary algorithms. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pp. 2585–2592, 2014. doi: 10.1109/CEC.2014.6900618.

Koza, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, 1992. ISBN 9780262111706.

Lin, H., Ye, H., Feng, W., Huang, Q., Li, Y., Lim, H., Li, Z., Wang, X., Ma, J., Liang, Y., and Zou, J. Can language models discover scaling laws?, 2026. URL https://arxiv.org/abs/2507.21184.

Liu, S., Agarwal, S., Maheswaran, M., Cemri, M., Li, Z., Mang, Q., Naren, A., Boneh, E., Cheng, A., Pan, M. Z., et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Miller, J. F. and Thomson, P. Cartesian genetic programming. In Poli, R., Banzhaf, W., Langdon, W. B., Miller, J. F., Nordin, P., and Fogarty, T. C. (eds.), *Genetic Programming, Proceedings of EuroGP 2000*, volume 1802 of *Lecture Notes in Computer Science*, pp. 121–132. Springer, 2000. doi: 10.1007/978-3-540-46239-2_9.

Nomura, M., Watanabe, S., Akimoto, Y., Ozaki, Y., and Onishi, M. Warm starting cma-es for hyperparameter optimization, 2020. URL https://arxiv.org/abs/2012.06932.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J., Mehrabian, A., et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Poloczek, M., Wang, J., and Frazier, P. I. Warm starting bayesian optimization, 2016. URL https://arxiv.org/abs/1608.03585.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Scott, E. O. and De Jong, K. A. Multitask evolution with cartesian genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pp. 255–256. ACM, 2017. doi: 10.1145/3067695.3075615.

Sharma, A. OpenEvolve: An open-source evolutionary coding agent. https://github.com/algorithmicsuperintelligence/openevolve, 2025. GitHub repository.

Swersky, K., Snoek, J., and Adams, R. Multi-task bayesian optimization. In Burges, C., Bottou, L., Welling, M., Ghahramani, Z., and Weinberger, K. (eds.), *Advances in Neural Information Processing Systems*, volume 26. Curran Associates, Inc., 2013. URL https://proceedings.neurips.cc/paper_files/paper/2013/file/f33ba15effa5c10e873bf3842afb46a6-Paper.pdf.

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.

Weisstein, E. W. Heilbronn triangle problem. From MathWorld–A Wolfram Web Resource, 2026. URL https://mathworld.wolfram.com/HeilbronnTriangleProblem.html. Accessed: 2026-04-30.

Xu, Y., Li, W., Vaezipoor, P., Sanner, S., and Khalil, E. B. Llms and the abstraction and reasoning corpus: Successes, failures, and the importance of object-based representations. *arXiv preprint arXiv:2305.18354*, 2023.

Appendix

This appendix is organized into five parts. Appendix A gives pseudocode for EMO-STA, including the three archive-initialization variants. Appendix B provides expanded in-distribution results, representative evolution trajectories, and additional compute-allocation and OOD evaluations. Appendix C describes task-family construction, scoring, the COVID Deaths and ARC-AGI case studies, and limitations. Appendix D includes representative prompts and program-level examples.

A. EMO-STA Algorithm Pseudocode

Algorithm 1 summarizes EMO-STA. The shared phase is identical across the three adaptation variants; the only difference is how the task-local archive is initialized from the shared archive. We therefore color the three initialization choices separately: *STA Warmstart* (green), *STA Best-Shared* (blue), and *STA Best-Local* (orange).

Algorithm 1 EMO-STA (Shared-Then-Adapt)

Require: Task family $\mathfrak{T} = \{\mathcal{T}_1, \ldots, \mathcal{T}_K\}$, shared budget S, per-task adaptation budget A, variant $v \in \{\textsc{Warmstart}, \textsc{Best-Shared}, \textsc{Best-Local}\}$
Ensure: Task-specific solutions $\{p_i^\star\}_{i=1}^K$
Initialize shared archive $\mathcal{A}^{\text{shared}} \leftarrow \varnothing$
Shared evolution
for $t = 1, \ldots, S$ **do**
  Propose or edit a candidate program p using the LLM and shared-archive context
  for $i = 1, \ldots, K$ **do**
    Evaluate p on task $\mathcal{T}_i$ and cache the task score $s_i(p)$
  end for
  Compute the shared score

$$s_{\text{shared}}(p) \leftarrow \frac{1}{K} \sum_{i=1}^{K} s_i(p)$$

  Insert p into $\mathcal{A}^{\text{shared}}$ together with $\{s_i(p)\}_{i=1}^K$ and $s_{\text{shared}}(p)$
end for
Task-local adaptation
Projection. Π_i converts a shared archive entry into a task-local archive entry for $\mathcal{T}_i$, reusing cached task scores when available.
for $i = 1, \ldots, K$ **do**
  if $v = \textsc{Warmstart}$ **then**
    $\mathcal{A}_i^{\text{init}} \leftarrow \{\Pi_i(p) : p \in \mathcal{A}^{\text{shared}}\}$ {STA Warmstart}
  else if $v = \textsc{Best-Shared}$ **then**
    $p_{\text{bs}}^\star \leftarrow \arg\max_{p \in \mathcal{A}^{\text{shared}}} s_{\text{shared}}(p)$ {STA Best-Shared}
    $\mathcal{A}_i^{\text{init}} \leftarrow \{\Pi_i(p_{\text{bs}}^\star)\}$
  else
    $p_{i,\text{bl}}^\star \leftarrow \arg\max_{p \in \mathcal{A}^{\text{shared}}} s_i(p)$ {STA Best-Local}
    $\mathcal{A}_i^{\text{init}} \leftarrow \{\Pi_i(p_{i,\text{bl}}^\star)\}$
  end if
  Set the task-local archive $\mathcal{A}_i \leftarrow \mathcal{A}_i^{\text{init}}$
  for $a = 1, \ldots, A$ **do**
    Propose or edit a candidate program using the LLM and task-local archive context
    Evaluate on $\mathcal{T}_i$, score with $s_i(\cdot)$, and update $\mathcal{A}_i$
  end for
  $p_i^\star \leftarrow \arg\max_{p \in \mathcal{A}_i} s_i(p)$
end for
return $\{p_i^\star\}_{i=1}^K$

B. Additional Results

B.1. Detailed In-Distribution Results

Tables 3 and 4 provide expanded versions of the main results, separating each method into its own row, including the pre-adaptation shared score, and reporting the *Shared / Adapt / Total* budget configuration for each family. *STA Best-Shared*

(Before Adaptation) denotes the best program produced by the shared evolution before any task-specific adaptation, so it is the initial program used by *STA Best-Shared*.

Table 3. Comparison of standard single-task and EMO-STA optimization for continuous optimization families. The budget row reports *Shared / Adapt / Total* iterations, where Total is computed as Shared plus the per-task adaptation budget times the number of tasks in the family.

Model	Method	Function minimization	Circle packing	Circle packing rectangles	Heilbronn triangle
Budget (Shared / Adapt / Total)		40/15/100	60/15/120	60/15/120	60/15/120
Haiku-4.5	STA Best-Shared (Before Adaptation)	$.887 \pm .06$	$.902 \pm .05$	$.832 \pm .01$	$.523 \pm .03$
	STA Best-Local	$\mathbf{.952 \pm .04}$	$.934 \pm .03$	$.861 \pm .02$	$\mathbf{.650 \pm .06}$
	STA Warmstart	$.949 \pm .05$	$.926 \pm .03$	$\mathbf{.865 \pm .02}$	$.628 \pm .05$
	STA Best-Shared	$.941 \pm .06$	$\mathbf{.940 \pm .02}$	$.845 \pm .01$	$.628 \pm .06$
	Single-task	$.888 \pm .05$	$.865 \pm .03$	$.832 \pm .01$	$.547 \pm .03$
Sonnet-4.5	STA Best-Shared (Before Adaptation)	$.862 \pm .03$	$.938 \pm .03$	$.875 \pm .04$	$.472 \pm .08$
	STA Best-Local	$\mathbf{.925 \pm .02}$	$\mathbf{.965 \pm .02}$	$\mathbf{.898 \pm .03}$	$\mathbf{.622 \pm .04}$
	STA Warmstart	$.917 \pm .02$	$.964 \pm .02$	$.890 \pm .03$	$.596 \pm .05$
	STA Best-Shared	$.904 \pm .03$	$.947 \pm .03$	$.892 \pm .04$	$.619 \pm .05$
	Single-task	$.891 \pm .05$	$.927 \pm .02$	$.840 \pm .02$	$.548 \pm .04$
Opus-4.5	STA Best-Shared (Before Adaptation)	$.877 \pm .07$	$.901 \pm .01$	$.935 \pm .01$	$.608 \pm .05$
	STA Best-Local	$\mathbf{.969 \pm .03}$	$\mathbf{.940 \pm .01}$	$\mathbf{.951 \pm .01}$	$\mathbf{.741 \pm .03}$
	STA Warmstart	$.942 \pm .07$	$.926 \pm .01$	$.943 \pm .01$	$.732 \pm .04$
	STA Best-Shared	$.941 \pm .09$	$.930 \pm .01$	$.943 \pm .01$	$.704 \pm .04$
	Single-task	$.914 \pm .05$	$.912 \pm .01$	$.912 \pm .01$	$.622 \pm .06$
Sonnet-4.6	STA Best-Shared (Before Adaptation)	$.946 \pm .03$	$.995 \pm .00$	$.993 \pm .00$	$.711 \pm .05$
	STA Best-Local	$.988 \pm .02$	$\mathbf{.997 \pm .00}$	$\mathbf{.986 \pm .01}$	$.862 \pm .04$
	STA Warmstart	$.973 \pm .03$	$\mathbf{.997 \pm .00}$	$.985 \pm .01$	$.809 \pm .04$
	STA Best-Shared	$\mathbf{.991 \pm .02}$	$\mathbf{.997 \pm .00}$	$.985 \pm .01$	$\mathbf{.865 \pm .07}$
	Single-task	$.901 \pm .02$	$.957 \pm .03$	$.967 \pm .02$	$.678 \pm .05$
Opus-4.6	STA Best-Shared (Before Adaptation)	$.942 \pm .04$	$.960 \pm .02$	$.941 \pm .01$	$.784 \pm .03$
	STA Best-Local	$\mathbf{.945 \pm .03}$	$\mathbf{.984 \pm .01}$	$\mathbf{.967 \pm .01}$	$.863 \pm .03$
	STA Warmstart	$.943 \pm .03$	$.972 \pm .02$	$.957 \pm .01$	$.844 \pm .03$
	STA Best-Shared	$.932 \pm .04$	$.979 \pm .02$	$.962 \pm .01$	$\mathbf{.877 \pm .03}$
	Single-task	$.895 \pm .04$	$.963 \pm .01$	$.944 \pm .01$	$.744 \pm .04$

B.2. Representative EMO-STA Trajectories

We include representative trajectories to show how shared evolution and task-local adaptation contribute over time. These examples are not intended to be exhaustive; they illustrate distinct adaptation modes, from light geometric calibration to broad task-specific improvement and targeted correction of a weak subtask. The *Shared / Adapt / Total* column reports $S/A/$Total, where S is the shared budget, A is the per-task adaptation budget, and Total $= S + KA$ is the matched family-level compute.

Discussion. The trajectories in Table 5 illustrate two complementary benefits of EMO-STA. In the circle-packing examples, shared evolution already discovers a reusable geometric solver scaffold, and adaptation mainly calibrates it to the selected task sizes or rectangle geometry. In the Heilbronn triangle example, adaptation has a larger effect, improving all subtasks and raising the family average far above the matched single-task baseline. The signal-processing example shows a more targeted correction: the shared program is already competitive on several signal types, while adaptation mainly repairs the step-change task, raising it from $.694$ to $.883$ and lifting the family average above the single-task reference. Overall, these trajectories suggest that EMO-STA helps both when shared evolution finds a strong general program that needs light retuning and when adaptation must make a focused task-local correction to a shared scaffold.

B.3. Additional Out-of-Distribution Evaluation Results

We include the full OOD holdout results for the three geometric optimization families. All programs are selected by their in-distribution runs and then evaluated on held-out task sizes without rerunning evolution.

Discussion on the additional results. As shown in Figures 10 to 12, held-out transfer is robust across compute-allocation

Table 4. Comparison of standard single-task and EMO-STA optimization for modeling and algorithmic optimization families. The budget row reports *Shared / Adapt / Total* iterations, where Total is computed as Shared plus the per-task adaptation budget times the number of tasks in the family.

Model	Method	Signal processing	SLDBench-3D	Rust adaptive sort	K-module
Budget (Shared / Adapt / Total)		60/10/100	60/10/80	60/10/100	40/20/120
Haiku-4.5	STA Best-Shared (Before Adaptation)	$.568 \pm .04$	$.936 \pm .02$	$.509 \pm .03$	$.392 \pm .05$
	STA Best-Local	$\mathbf{.600 \pm .05}$	$\mathbf{.958 \pm .02}$	$.533 \pm .02$	$.567 \pm .06$
	STA Warmstart	$.584 \pm .06$	$.953 \pm .02$	$.535 \pm .02$	$.567 \pm .04$
	STA Best-Shared	$.597 \pm .04$	$.949 \pm .02$	$.509 \pm .03$	$\mathbf{.575 \pm .07}$
	Single-task	$.569 \pm .01$	$.951 \pm .01$	$\mathbf{.539 \pm .02}$	$.550 \pm .03$
Sonnet-4.5	STA Best-Shared (Before Adaptation)	$.559 \pm .02$	$.955 \pm .02$	$.458 \pm .03$	$.367 \pm .02$
	STA Best-Local	$\mathbf{.587 \pm .01}$	$\mathbf{.976 \pm .01}$	$.481 \pm .03$	$.617 \pm .03$
	STA Warmstart	$.578 \pm .02$	$.971 \pm .01$	$.484 \pm .03$	$\mathbf{.650 \pm .02}$
	STA Best-Shared	$.582 \pm .02$	$.971 \pm .02$	$.457 \pm .03$	$.567 \pm .06$
	Single-task	$.576 \pm .01$	$.959 \pm .01$	$\mathbf{.528 \pm .01}$	$.617 \pm .05$
Opus-4.5	STA Best-Shared (Before Adaptation)	$.612 \pm .03$	$.959 \pm .02$	$.483 \pm .05$	$.442 \pm .02$
	STA Best-Local	$.620 \pm .03$	$\mathbf{.983 \pm .00}$	$.515 \pm .05$	$.617 \pm .03$
	STA Warmstart	$\mathbf{.635 \pm .03}$	$.972 \pm .01$	$\mathbf{.520 \pm .05}$	$\mathbf{.675 \pm .03}$
	STA Best-Shared	$.625 \pm .02$	$.981 \pm .00$	$.483 \pm .05$	$.592 \pm .03$
	Single-task	$.568 \pm .01$	$.973 \pm .01$	$.497 \pm .02$	$.567 \pm .05$
Sonnet-4.6	STA Best-Shared (Before Adaptation)	$.607 \pm .05$	$.959 \pm .01$	$.656 \pm .01$	$.383 \pm .05$
	STA Best-Local	$\mathbf{.628 \pm .04}$	$\mathbf{.969 \pm .01}$	$.659 \pm .01$	$.617 \pm .09$
	STA Warmstart	$.626 \pm .04$	$.968 \pm .01$	$\mathbf{.663 \pm .01}$	$\mathbf{.700 \pm .07}$
	STA Best-Shared	$.613 \pm .05$	$\mathbf{.969 \pm .01}$	$.656 \pm .01$	$.575 \pm .03$
	Single-task	$.608 \pm .03$	$.955 \pm .01$	$.616 \pm .03$	$.675 \pm .05$
Opus-4.6	STA Best-Shared (Before Adaptation)	$.653 \pm .04$	$.958 \pm .02$	$.612 \pm .02$	$.450 \pm .03$
	STA Best-Local	$.713 \pm .05$	$\mathbf{.975 \pm .01}$	$.616 \pm .02$	$.725 \pm .02$
	STA Warmstart	$.707 \pm .04$	$.973 \pm .01$	$\mathbf{.625 \pm .02}$	$\mathbf{.800 \pm .05}$
	STA Best-Shared	$\mathbf{.716 \pm .04}$	$.967 \pm .01$	$.612 \pm .02$	$.692 \pm .05$
	Single-task	$.648 \pm .03$	$.964 \pm .02$	$.531 \pm .05$	$.758 \pm .08$

Example	Model / method	Shared / Adapt / Total	Shared	Adapt	Single-task
Circle packing	Haiku-4.5, STA Best-Local	60/15/120	.833	$.903 \rightarrow .925$	.865
Circle-packing rectangles	Sonnet-4.5, STA Best-Shared	60/15/120	.894	$.894 \rightarrow .924$	.840
Heilbronn triangle	Sonnet-4.6, STA Best-Shared	60/15/120	.750	$.750 \rightarrow .905$	.678
Signal processing	Opus-4.6, STA Best-Local	60/10/100	.619	$.635 \rightarrow .685$	.648

Table 5. Representative EMO-STA trajectory examples. The Shared column reports the final family-average score at the end of shared evolution. The Adapt column reports the average score at the start and end of task-local adaptation. Single-task reports the mean over five independent single-task runs for the same family, model, and matched family-level total compute.

settings, but that the three adaptation strategies have different OOD profiles. Across all three geometric families, EMO-STA variants generally outperform the direct single-task baseline on held-out task sizes, indicating that the shared phase learns reusable geometric structure rather than only improving the in-distribution tasks. *STA Best-Shared* is the most reliable OOD strategy overall, especially for circle packing in rectangles and Heilbronn triangle, where the globally best shared program transfers better than task-specialized adaptations across most held-out sizes. This suggests that, when the held-out task may differ from any one training task, the shared-average solution can preserve more family-level robustness. *STA Warmstart* is particularly competitive for unit-square circle packing, where retaining a diverse archive of layouts helps across neighboring circle counts. *STA Best-Local* remains strong in several settings, but its advantage is more tied to proximity between the adaptation task and the held-out size, so it is most useful when the target task is close to the evaluation size. Together, these patterns indicate that shared evolution improves transfer beyond the training task sizes.

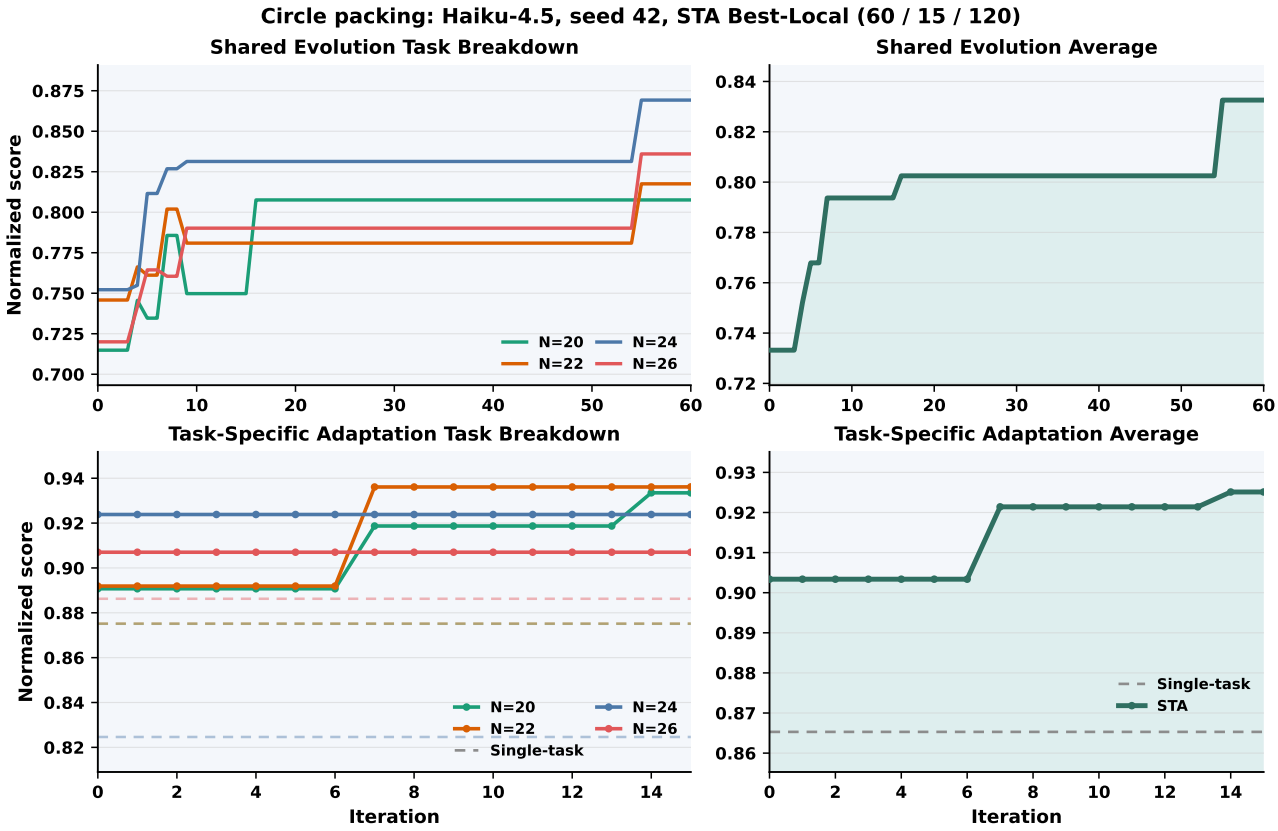


Figure 6. Circle-packing trajectory for Haiku-4.5, seed 42, using *STA Best-Local* with a $60/15/120$ $S/A/\text{Total}$ setting. The adapted family average increases from .903 to .925, compared with a five-run Single-task average of .865.

B.4. Additional Compute Allocation Results

We report additional compute-allocation results for Heilbronn triangle and the two circle-packing families, complementing the main-text results on Heilbronn triangle and function minimization.

Discussion on the additional results. As shown in Figures 13 to 15, shared evolution remains beneficial across both ways of varying compute: increasing the total budget after a fixed shared phase, and changing the shared/adaptation split under a fixed total budget. For Heilbronn triangle, increasing the total budget improves the direct single-task baseline, but the EMO-STA variants remain consistently stronger; the gap is largest at lower budgets, where the fixed shared phase provides useful geometric structure before task-specific adaptation. The two circle-packing families show the same qualitative advantage under fixed total compute. The unit-square circle-packing family benefits most from a balanced allocation, with the strongest result at $60/15/120$, while circle packing in rectangles is less sensitive to the exact allocation, suggesting that shared evolution quickly discovers reusable geometry-aware structure and adaptation mainly refines it for each task size and rectangle aspect ratio. Across these results, *STA Best-Local* is usually strongest or competitive, while *STA Warmstart* and *STA Best-Shared* also remain above the single-task baseline, reinforcing that the gain comes from shared evolution rather than a single initialization choice.

C. Experimental Details

C.1. Details on Task-Family Construction, Scoring, and Compute Details

For every task family, we converted an existing OpenEvolve example or closely related benchmark setup into a small collection of related subtasks that share one evolving artifact, one evaluator family, and the same shared-then-adapt workflow (Sharma, 2025). In the shared phase, the evolving program is optimized against the average score across the family. We then initialize task-specific continuations from the shared archive and compare them against direct single-task baselines

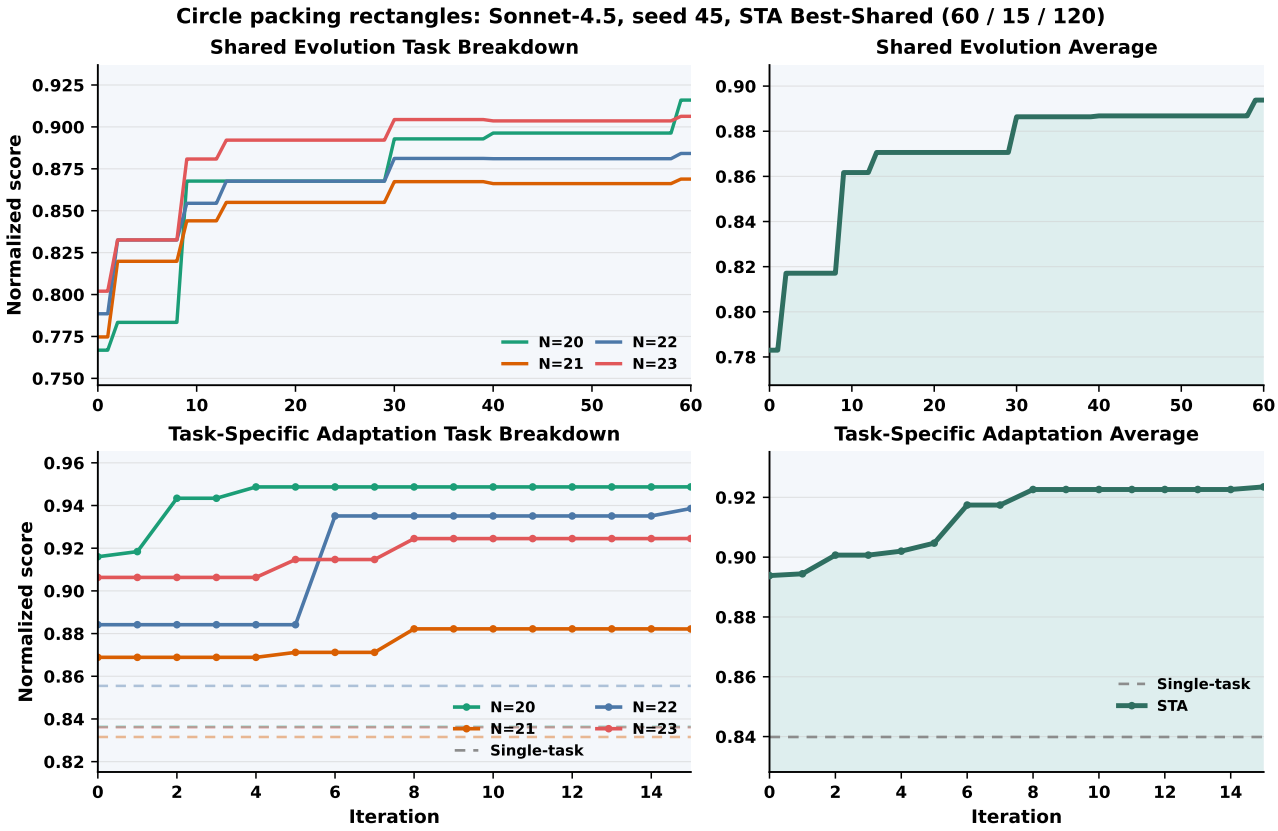


Figure 7. Circle-packing-rectangles trajectory for Sonnet-4.5, seed 45, using *STA Best-Shared* with a $60/15/120$ $S/A/\text{Total}$ setting. Adaptation improves the average score from .894 to .924, above the five-run Single-task average of .840.

with matched interfaces and budgets. The main design goal is to expose reusable structure that can be discovered during shared optimization, while still leaving enough task-specific variation for adaptation to matter. Below, we first describe shared implementation and compute details, then give the task-specific construction and scoring rules for each benchmark family.

C.1.1. Shared implementation and compute details

OpenEvolve configuration. The underlying OpenEvolve hyperparameters and run configurations mostly follow the original OpenEvolve example settings, while EMO-STA introduces task-family-specific interfaces, evaluators, and shared/adaptation budget schedules. The exact configuration files used for each experiment are included in the supplementary code.

LLM access. Claude-family models were accessed through Amazon Bedrock, and Gemini models were accessed through the Gemini Developer API via Google AI Studio rather than Vertex AI. Full model identifiers and sampling parameters are provided in the supplementary code.

C.1.2. Task families

Function minimization. We adapted the original standalone function-minimization example from OpenEvolve (Sharma, 2025) into a four-task family of public two-dimensional objectives commonly used in global-optimization benchmarks (Jamil & Yang, 2013).

- **Oscillatory Basin** (original example) preserves the spirit of the original sinusoidal landscape, with smooth periodic structure and multiple local basins.
- **Ackley** adds a broad, nearly flat outer region with a sharp central basin.

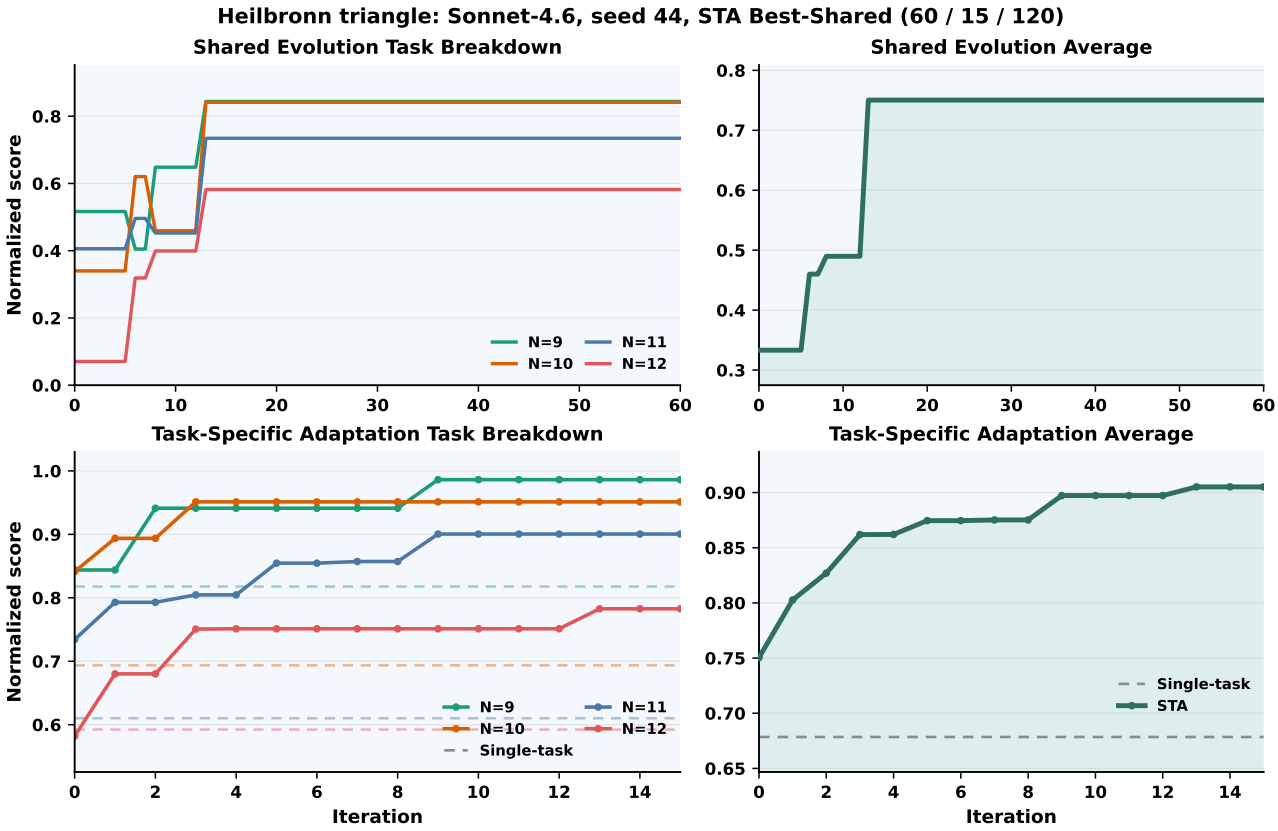


Figure 8. Heilbronn-triangle trajectory for Sonnet-4.6, seed 44, using *STA Best-Shared* with a $60/15/120$ $S/A/\text{Total}$ setting. This example shows broad task-specific improvement, with the average score increasing from .750 to .905 versus a five-run Single-task average of .678.

- **Rastrigin** introduces a highly multimodal landscape with many regularly spaced local optima.
- **Rosenbrock** tests narrow-valley optimization with strong variable coupling.

Rather than allowing the evolving code to adapt to a single named landscape, the EMO-STA version requires one generic derivative-free optimizer that receives only an opaque `objective_fn` and `bounds` from the evaluator. The benchmark functions are translated but not rescaled, and the task name and optimum are hidden from the candidate. Bounds are $[-5, 5]^2$ for Oscillatory Basin and Ackley, $[-5.12, 5.12]^2$ for Rastrigin, and $[-3, 3]^2$ for Rosenbrock. The full evaluator calls each candidate with `iterations=200` and five deterministic seeds $(0, \ldots, 4)$, while the cheaper cascade stage uses `iterations=50` and two seeds. For each seed, the evaluator recomputes the true objective value at the returned point and rejects points outside bounds. The task score is

$$0.50(1 + \Delta f)^{-1} + 0.35(1 + d)^{-1} + 0.15\rho,$$

where Δf is the nonnegative gap to the task optimum, d is Euclidean distance to the translated optimum, and ρ is the successful-trial fraction.

Signal processing. We adapted the original OpenEvolve signal-processing benchmark, which evaluated a single algorithm across several synthetic signal types, into four explicit EMO-STA subtasks (Sharma, 2025):

- **Trend+sine** combines a smooth global trend with a periodic component.
- **Multifrequency** superposes multiple sinusoidal components at different frequencies.
- **Chirp** uses a sinusoid whose frequency changes over time.

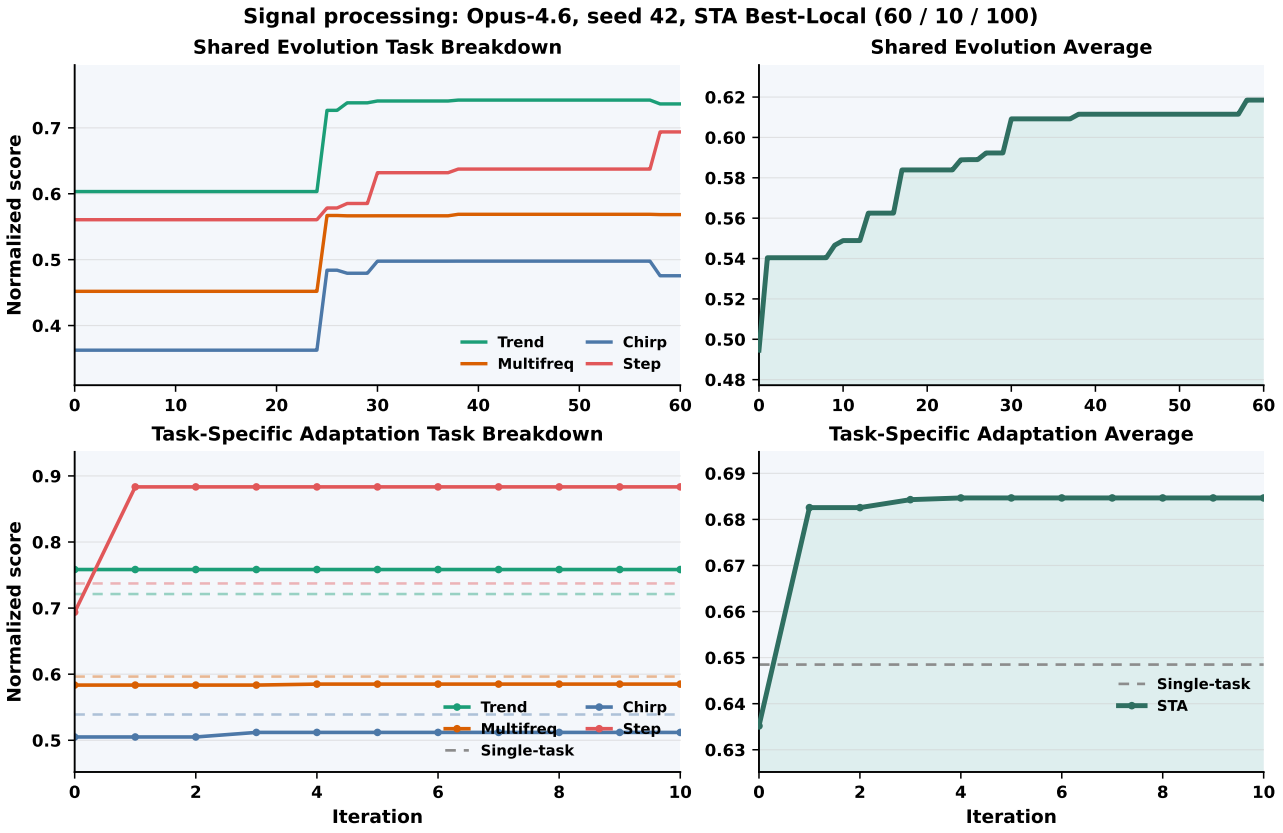


Figure 9. Signal-processing trajectory for Opus-4.6, seed 42, using *STA Best-Local* with a $60/10/100$ $S/A/\text{Total}$ setting. Adaptation is concentrated on the step-change task, which improves from .694 to .883, raising the family average from .635 to .685 above the five-run Single-task average of .648.

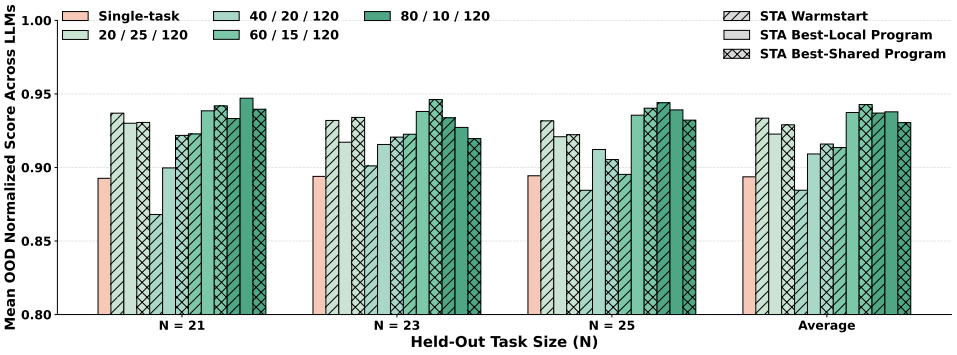


Figure 10. OOD holdout evaluation for circle packing across EMO-STA budget allocations with the single-task baseline fixed at 120 total iterations. The x-axis shows held-out task sizes plus the average across holdouts. The peach bars show the fixed single-task baseline, green colors denote the *Shared / Per-task adaptation / Total* budget allocation, and hatch patterns denote the STA adaptation variant. Bars report mean OOD normalized score across LLMs.

- **Step changes** contains abrupt piecewise shifts in the underlying signal level.

All tasks use the same causal interface, `process_signal(noisy_signal, window_size)`, with `window_size` $= 20$. The task lengths are $500, 600, 700, 800$, with Gaussian noise standard deviations $0.2, 0.3, 0.4, 0.5$, respectively. Full evaluation uses three fixed noisy realizations per task, with seeds $0, 1, 2$, while the cascade stage uses seed 0. The clean signal and noisy observation are deterministic functions of the task and seed, so evaluation is fixed across runs. The candidate observes only the noisy input signal and must return a one-dimensional filtered signal of length

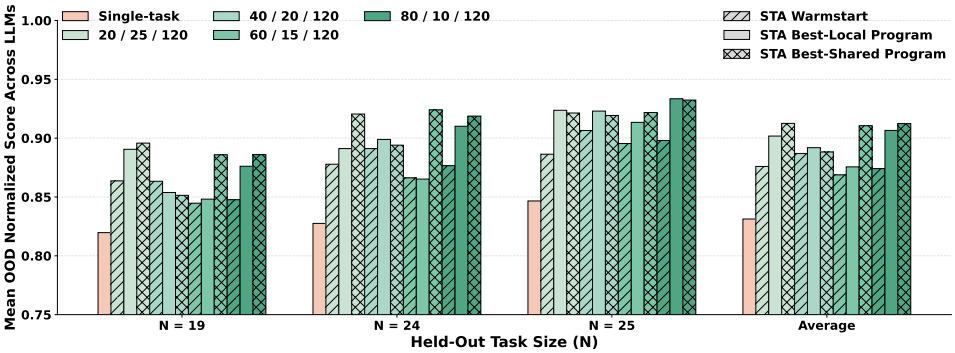


Figure 11. OOD holdout evaluation for circle packing in rectangles across EMO-STA budget allocations with the single-task baseline fixed at 120 total iterations. The x-axis shows held-out task sizes plus the average across holdouts. The peach bars show the fixed single-task baseline, green colors denote the *Shared / Per-task adaptation / Total* budget allocation, and hatch patterns denote the STA adaptation variant. Bars report mean OOD normalized score across LLMs.

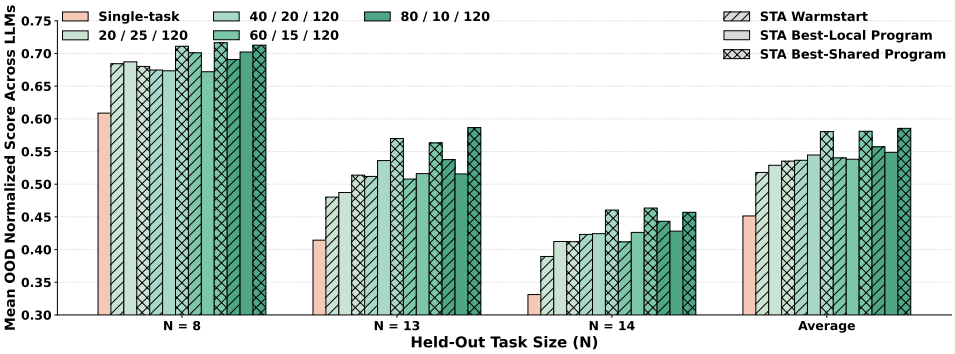


Figure 12. OOD holdout evaluation for the Heilbronn triangle task across EMO-STA budget allocations with the single-task baseline fixed at 120 total iterations. The x-axis shows held-out task sizes plus the average across holdouts. The peach bars show the fixed single-task baseline, green colors denote the *Shared / Per-task adaptation / Total* budget allocation, and hatch patterns denote the STA adaptation variant. Bars report mean OOD normalized score across LLMs.

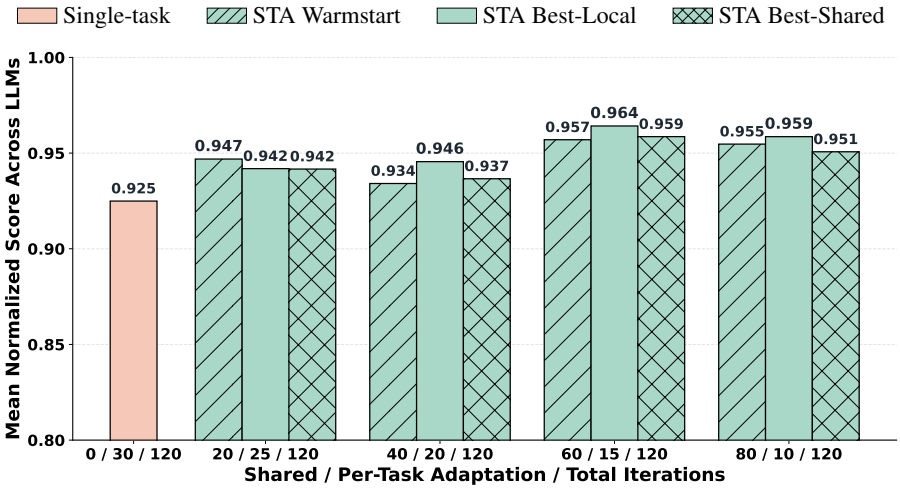


Figure 13. Compute-allocation results for EMO-STA on *circle packing*, with the single-task baseline fixed at $B = 30$ per task, corresponding to $KB = 120$ total iterations. Grouped bars show *STA Warmstart*, *STA Best-Local*, and *STA Best-Shared* under different *Shared / Per-task Adapt / Total* allocations. Here, *Shared* is the family-level shared budget S, *Adapt* is the per-task adaptation budget A, and *Total* is $S + KA$. The leftmost bar is the direct single-task baseline, with allocation $0/30/120$. Bars report the mean score averaged over the five models Claude Haiku-4.5, Sonnet-4.5, Sonnet-4.6, Opus-4.5, and Opus-4.6.

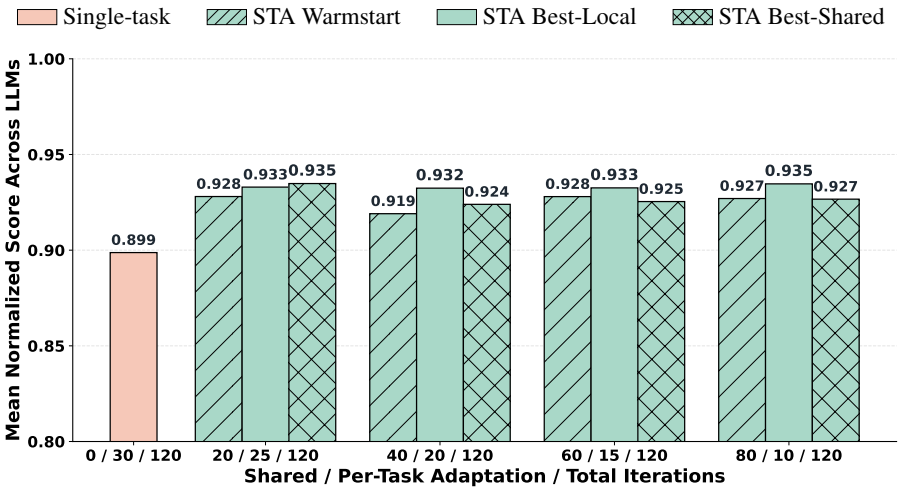


Figure 14. Compute-allocation results for EMO-STA on *circle packing in rectangles*, with the single-task baseline fixed at $B = 30$ per task, corresponding to $KB = 120$ total iterations. Grouped bars show *STA Warmstart*, *STA Best-Local*, and *STA Best-Shared* under different *Shared / Per-task Adapt / Total* allocations. Here, *Shared* is the family-level shared budget S, *Adapt* is the per-task adaptation budget A, and *Total* is $S + KA$. The leftmost bar is the direct single-task baseline, with allocation $0/30/120$. Bars report the mean score averaged over the five models Claude Haiku-4.5, Sonnet-4.5, Sonnet-4.6, Opus-4.5, and Opus-4.6.

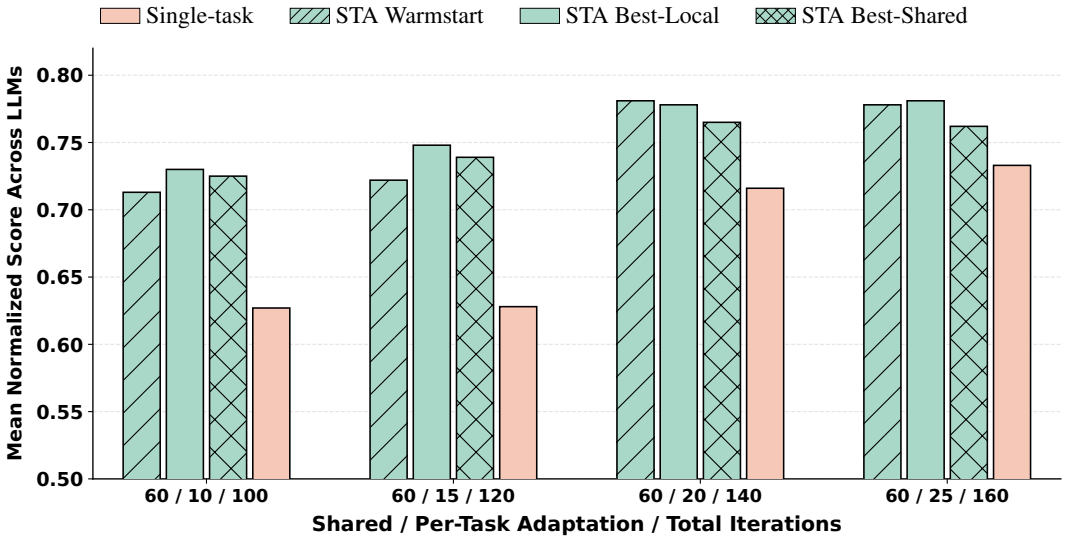


Figure 15. Compute-allocation results for the Heilbronn triangle family with the shared budget fixed at $S = 60$. The x-axis reports *Shared / Per-task Adapt / Total* iterations. For each total budget, the direct single-task baseline uses the corresponding matched per-task budget B, so that $S + KA = KB$. Bars report mean normalized score across LLMs and seeds for *STA Warmstart*, *STA Best-Local*, *STA Best-Shared*, and direct single-task optimization.

`len(noisy)` $-$ `window_size` $+ 1$; it never sees the clean target, task identifier, or generating formula. The evaluator measures slope reversals, recent lag error, average tracking error against the aligned noisy signal, false reversals against the clean signal, correlation with the clean signal, and noise reduction. The final task score is a normalized composite in $[0, 1]$, combining denoising quality, smoothness, clean-signal correlation, noise reduction, and success rate; failures or timeouts receive zero. We excluded the random-walk case from the EMO-STA family so that the shared tasks remain closely related while still exhibiting distinct denoising and trend-recovery behavior.

Circle packing. We adapted the unit-square circle-packing benchmark with fixed $n = 26$, used in AlphaEvolve and implemented in OpenEvolve, into a task family with similar circle counts (Novikov et al., 2025; Sharma, 2025). The EMO-STA training tasks use $n \in \{20, 22, 24, 26\}$, all through a single `construct_packing(n)` or `run_packing(n)` program interface, so the evolved code must implement a reusable packing strategy rather than adapt to one fixed problem

size. The evaluator expects centers of shape $(n, 2)$ and radii of shape $(n,)$, and independently validates finite nonnegative radii, containment in $[0, 1]^2$, and pairwise non-overlap with tolerance 10^{-6}. Invalid packings, execution failures, and timeouts receive score zero. Valid packings are scored by the normalized ratio `sum_radii/target_sum_radii`, using targets $2.301, 2.420, 2.530, 2.635$ for $n = 20, 22, 24, 26$, taken from public best-known circle-packing tables (Friedman, 2026b). The ratio is not capped, so a packing exceeding the reference can score above one. We also define evaluation-only holdouts at $n \in \{21, 23, 25\}$, with targets $2.362, 2.478, 2.587$, to test whether the shared representation transfers to unseen but nearby circle counts. This normalization is necessary because absolute radius sums are not directly comparable across different n: larger circle counts naturally allow larger total sums, so averaging raw sums would bias shared evolution toward the larger-n tasks.

Circle packing in rectangles. We also define a second circle-packing EMO-STA family that keeps the same shared constructive structure but changes the container geometry to the perimeter-4 rectangle setting (Friedman, 2026a). In this family, the evolving program still implements `construct_packing(n)` or `run_packing(n)`, but it must return centers, radii, and a rectangle width α. The evaluator sets the height to $2 - \alpha$, so the rectangle perimeter is 4, and requires $0 < \alpha \leq 1$, finite nonnegative radii, containment in $[0, \alpha] \times [0, 2 - \alpha]$, and pairwise non-overlap. The four public training tasks use $n \in \{20, 21, 22, 23\}$, with target sums $2.305, 2.365, 2.425, 2.484$. Valid packings are scored by the uncapped ratio between the summed radius and the task-specific target. Evaluation-only OOD rectangle tasks use $n \in \{19, 24, 25\}$, with targets $2.241, 2.535, 2.592$. As in unit-square circle packing, normalization is essential because attainable absolute sums vary with both n and the geometry of the best rectangle, so raw summed radii do not provide a useful family-level signal for shared evolution.

Heilbronn triangle. We adapted the Heilbronn triangle benchmark into a four-task EMO-STA family over nearby point counts inside one fixed canonical unit-area triangle with vertices $(0, 0)$, $(2, 0)$, and $(0, 1)$, using known small-n reference values for normalization (Weisstein, 2026). The evolving code uses a generic `construct_points(n)` or `run_heilbronn(n)` interface and must maximize the minimum triangle area induced by all triples of points. The evaluator validates shape $(n, 2)$, finiteness, and containment in the canonical triangle, and then exactly computes the minimum area over all point triples. Invalid outputs receive score zero. Valid outputs are scored as `min_triangle_area/target_min_area`, using references $0.0548469, 0.0433767, 0.0360927, 0.0310048$ for $n = 9, 10, 11, 12$. Evaluation-only OOD tasks use $n \in \{8, 13, 14\}$, with references $0.0677891, 0.0245643, 0.0237758$. The normalization is important because the attainable optimum changes substantially with n: as more points are packed into the same canonical triangle, the best achievable minimum area decreases, so averaging raw areas would bias the shared objective toward easier smaller-n tasks. The normalized objective instead puts the training tasks on a common scale and encourages the shared phase to learn reusable geometric placement structure across nearby sizes.

K-module. We adapted the original public OpenEvolve 4-module, 5-option K-module problem into a harder hidden-family EMO-STA benchmark (Sharma, 2025). The EMO-STA version uses six named modules with six opaque options each, giving 6^6 possible configurations, and defines four hidden target tasks. A candidate must return one complete configuration through `run_pipeline()` or `configure_pipeline()`; malformed configurations or invalid option names receive zero. Each hidden task has a fixed target configuration, and the task score is the fraction of modules matched, `correct_modules/6`. Thus, a task-local score of $2/3$ means four of six modules match the hidden target. The four hidden targets are deterministic and fixed across all seeds and runs. They are included in the supplementary evaluator code for reproducibility, but the prompts and public artifacts expose only the module names and option counts. The targets are constructed so that each task agrees with the shared consensus configuration on exactly three of six modules, creating a shared optimum that is useful but not identical to any one task-specific optimum. This forces the evolving code to learn a generic compositional strategy that can later be retuned for one specific hidden target.

SLDBench-3D. We adapted SLDBench, a benchmark for automated scaling-law discovery, into a two-task EMO-STA subset containing `vocab_scaling_law` and `data_constrained_scaling_law` (Lin et al., 2026). Both tasks are loaded from `pkuHaowei/sldbench` and use the provided train/test splits. They are 3D scalar-loss scaling-law tasks with a 7-parameter cap, making them close enough for transfer while still non-identical. We canonicalize both tasks to the same three-column schema, [`model_size_like`, `diversity_like`, `total_data_like`]: vocabulary scaling uses [`non_vocab_parameters`, `vocab_size`, `num_characters`], while data-constrained scaling maps `params`, `unique_tokens`, and `tokens` into the canonical schema. For each group, the evaluator fits coefficients on the train split by calling `fit_scaling_law`, and then evaluates `scaling_law_func` on the corresponding held-out test group. The fitted parameter vector must contain between one and seven finite parameters, so EMO-STA shares the law form and fitting routine but refits coefficients locally for each group. The score is $1/(1 + \text{NMSE})$, where NMSE is test MSE normalized by

the variance of the held-out targets. This preserves the intended scaling-law structure while making the family amenable to shared optimization.

Rust adaptive sort. We adapted the original standalone OpenEvolve Rust sorting benchmark into four deterministic regimes: random, nearly sorted, reverse sorted, and duplicates (Sharma, 2025). The EMO-STA evaluator copies each candidate into a temporary Cargo project, compiles it once in release mode, and then benchmarks the compiled binary on the selected task regime, so one `adaptive_sort` implementation is reused across all tasks. Compile failures, runtime failures, malformed output, and timeouts receive zero. Random, nearly sorted, and duplicate tasks use array sizes 1000 and 10000 with seeds $0, 1, 2$; reverse-sorted tasks use the same sizes without random seeds. Nearly sorted arrays use 5% random swaps, and duplicate arrays use 10 unique values at size 1000 and 100 unique values at size 10000. Each dataset is checked for exact sorted correctness against Rust `sort_unstable`; any task with less than 100% correctness receives score zero. For correct tasks, the evaluator compares candidate median runtime to the median runtime of `sort_unstable`, measured in the same binary after one warmup and five benchmark repetitions. The score is $0.8s + 0.2c$, where $s = \texttt{mean_speedup}/(1 + \texttt{mean_speedup})$ rewards speed and $c = 1/(1 + \mathrm{CV})$ rewards consistency across datasets. This forces the shared phase to discover broadly useful control logic for switching among sorting behaviors, while still allowing task-specific adaptation to calibrate the implementation to one regime. We intentionally omitted the original `partially_sorted` regime from the initial family and reserve it as a possible holdout or generalization check.

Discussion on compute matching and practical overhead. The EMO-STA variants are compute-matched to the single-task baseline at the level of evolution iterations, and the adaptation initializations add little extra overhead. During shared evolution, each shared candidate is evaluated on the task family, and these per-task scores are cached. Thus, *STA Warmstart* can initialize the task-local archive by transferring the shared programs together with their cached task scores, while *STA Best-Local* can select the target-task best shared candidate directly from the cached scores before continuing adaptation. In practice, the dominant cost in LLM-guided evolution is usually the LLM call used to generate each new candidate program, rather than archive transfer or additional evaluator-side evaluation. Matching the number of evolution iterations therefore provides a reasonable compute comparison between shared-then-adapt search and independent single-task runs.

C.2. ARC-AGI Case Study Details

Using the ARC-AGI example from the OpenEvolve repository as our implementation base (Sharma, 2025), we evaluate EMO-STA on two model-specific sets of failed single-task ARC runs from the ARC Prize 2025 ARC-AGI evaluation split, selecting the first 20 failed run–task instances in evaluation-split order for Gemini-3.1-Pro-Preview and 20 for Claude Opus 4.6. This appendix describes how we construct transformed task families, match budgets, and diagnose overfitting in the single-task baseline.

Task-family construction. For each base ARC task, we construct a five-task augmented family consisting of the original task, a 90° rotation, a left-right flip, an up-down flip, and a fixed color permutation, following Akyürek et al. (2024). Each transformation is applied consistently to every input grid and every target output grid, so the surface representation changes while the underlying reasoning rule is preserved up to the corresponding transformation. For transformed variants, exact-match evaluation is performed in the transformed task representation. Thus, the augmented family provides related task diversity without using the held-out test output for optimization or introducing task-specific solution templates.

Color permutation. ARC colors are integer labels from 0 to 9, so a color transformation can be implemented as a permutation of these labels. We use the fixed remapping

```
permutation = np.array([0, 2, 3, 4, 5, 6, 7, 8, 9, 1], dtype=np.int64)
return permutation[np.asarray(grid, dtype=np.int64)]
```

This leaves color 0 unchanged, which is useful because ARC often uses 0 as the background color, and cyclically remaps the remaining colors. Since the same remapping is applied to both inputs and outputs, the task logic is unchanged; only the color names differ.

Evolution objective and OpenEvolve configuration. All ARC runs use diff-based program evolution with seed 42 and either Gemini-3.1-Pro-Preview or Claude Opus 4.6 as the primary model. Programs are optimized using ARC train-set pass@2: each program emits two candidate output grids for each training input, and an example is counted as solved if either candidate exactly matches the target grid. Held-out test grids are used only for evaluation and overfitting diagnosis, not for optimization. Unless otherwise specified, the evolution configuration uses population size 60, archive size 30, four islands, MAP-Elites features over score and diversity with 10 bins, migration every 15 iterations, and migration rate 0.15.

The exact run configurations are included in the supplementary code.

Budget matching. For each augmented ARC family, EMO-STA first runs a shared evolutionary phase for 60 iterations across all five variants, then specializes separately to each variant for 15 iterations. The direct single-task baseline evolves each variant independently for 27 iterations. This matches the family-level iteration budget:

$$60 + 5 \times 15 = 5 \times 27 = 135.$$

Thus, the EMO-STA and single-task comparisons use the same total number of evolution iterations across the five related variants. We evaluate *STA Warmstart*, *STA Best-Shared*, and *STA Best-Local* under this same matched budget; the main-text solve rates are measured on the original base tasks after adaptation, using their held-out test inputs.

Failed-case selection and overfitting diagnosis. For each primary model, we first run the standard single-task evolution baseline on tasks from the ARC Prize 2025 ARC-AGI evaluation split using the same train-set pass@2 objective. A run is marked as failed if its best evolved program does not solve the held-out test input. Because evolutionary search is stochastic, failure is defined at the run level rather than as a permanent property of a task; when needed, we repeat the same evaluation protocol to obtain enough failed runs. We then select the first 20 failed run–task instances per model in evaluation-split order, separately for Gemini-3.1-Pro-Preview and Claude Opus 4.6, and use the corresponding base tasks for the augmented multi-task experiments. To diagnose overfitting, we inspect whether the best program from each failed run nevertheless solves all provided training examples under pass@2. When this happens, we classify the failure as overfitting: the program fits the sparse training pairs but fails to generalize to the held-out test grid.

The exact sampled failed run–task instances are listed in Table 6.

Gemini-3.1-Pro-Preview		Claude Opus 4.6	
Dataset index	ARC task ID	Dataset index	ARC task ID
0	0934a4d8	0	0934a4d8
5	16b78196	5	16b78196
8	195c6913	6	16de56c4
9	1ae2feb7	9	1ae2feb7
11	20a9e565	10	20270e3b
14	247ef758	11	20a9e565
18	291dc1e1	14	247ef758
41	581f7754	18	291dc1e1
46	62593bfd	28	3a25b0d8
52	6ffbe589	38	4e34c42c
63	800d221b	41	581f7754
67	88e364bc	58	7b3084d4
69	898e7135	69	898e7135
70	8b7bacbf	70	8b7bacbf
90	b5ca7ac4	82	a32d8b75
98	cbebaa4b	85	a6f40cea
103	db0c5428	90	b5ca7ac4
106	dd6b8c4b	103	db0c5428
110	e3721c99	108	dfadab01
115	eee78d87	110	e3721c99

Table 6. Selected failed ARC run–task instances used in the ARC-AGI case study. Each entry is reported as the task's position in the ARC Prize 2025 ARC-AGI evaluation split and its ARC task ID. The selected cases are the first 20 observed failures for each model in evaluation-split order; rows are paired only for compact presentation and do not imply correspondence between the two model-specific sets.

C.3. COVID Deaths Time-Series Case Study

We use the EFE-Time setting of concurrent anonymized work (Authors, 2026) to study overfitting in evolutionary optimization for data-science tasks. EFE-Time evolves time-series preprocessing programs rather than forecasting models. Each candidate implements `fit`, `transform`, and `inverse_transform`: the program is fit on the available history, transforms the input series, passes the transformed sequence to a fixed Chronos-2 forecaster (Ansari et al., 2025), and maps the forecast back to the original scale before evaluation. The evolutionary objective is therefore downstream forecasting performance, which makes this setting a natural stress test for overfitting: with short or noisy series, evolution can select transformations that exploit validation windows without improving held-out forecasts.

The dataset is GIFT-Eval's COVID Deaths task, a daily healthcare benchmark with 266 univariate series and a 30-step

prediction horizon (Aksu et al., 2024). The target is confirmed COVID-19 deaths. These series are heterogeneous across countries and noisy due to underreporting, reporting delays, day-of-week effects, and differences in reporting conventions. Thus, a transformation evolved separately for one series may fit local reporting artifacts rather than reusable forecasting structure.

In our EMO formulation, each COVID Deaths series is treated as one task. The single-task baseline runs EFE-Time independently for each series, evolving a separate transformation from scratch. STA Best-Shared instead performs one shared evolution over the full family of 266 series, selects the transformation with the best family-level objective, and applies it directly to each series with no task-local adaptation. This corresponds to the native EFE-Time no-adaptation setting from Authors (2026). To match evolutionary budgets, we run each single-task evolution for 5 iterations, giving $266 \times 5 = 1330$ total iterations, and run the shared STA Best-Shared evolution for 1330 iterations.

The results in Figure 5a indicate that the single-task baseline overfits. It improves validation MASE and wQL by 12.32% and 12.99%, respectively, but these validation gains largely vanish on held-out test windows: test MASE worsens by 1.12% and test wQL improves by only 2.71%. STA Best-Shared without adaptation obtains smaller or comparable validation gains but much stronger test gains, improving test MASE by 13.24% and test wQL by 32.56%. This pattern supports the regularization view of shared evolution: because the selected transformation must perform well across many noisy, heterogeneous series, it is less likely to encode idiosyncratic artifacts from any single series. In preliminary runs, adding task-local adaptation after the shared phase could reintroduce overfitting on this dataset, so we report the no-adaptation STA Best-Shared variant for this case study.

C.4. Scope and Limitations

EMO-STA is designed for task families that can be expressed through a shared executable interface, and is therefore most appropriate when tasks share meaningful program structure. Our experiments focus on such integrable families, so future work could extend the evaluation to larger task collections, more general forms of task relatedness, and automated criteria for deciding when shared evolution should be applied.

D. Prompts and Program Examples

D.1. Sample Prompts for Shared Program Interfaces

We include representative prompts illustrating how each task family is framed around a common entry-point function while encouraging reusable solutions that transfer across subtasks.

Listing 1. Example system prompt for the EMO-STA circle-packing task family.

```
You are an expert mathematician specializing in circle packing and computational
    geometry.

Your task is to improve a generic constructor/optimizer for packing n circles of
    varying radii inside the unit square [0,1] x [0,1], maximizing the sum of their
    radii.

The evolving code must preserve these exact function signatures:

def construct_packing(n: int):
    ...
    return centers, radii, sum_radii

def run_packing(n: int):
    return construct_packing(n)

The evaluator will call your code for multiple related tasks with:
n in {20, 22, 24, 26}

Public reference target sums used for scoring:
- n=20: 2.301
- n=22: 2.420
- n=24: 2.530
- n=26: 2.635
```

```
Constraints:
- centers must have shape (n, 2)
- radii must have shape (n,)
- all circles must lie fully inside the unit square
- circles must not overlap
- keep all outputs finite
- keep the algorithm deterministic

Guidance:
- Favor reusable geometric construction and optimization routines parameterized by n
- Avoid implementing four completely separate hardcoded exact layouts as the main
    strategy
- Avoid implementing a lookup table of exact layouts only for the seen n values
- Explicit constructive geometry, symmetry-breaking, ring/grid hybrids, corner
    utilization, and local continuous optimization are all reasonable
- Computing radii from fixed centers is acceptable; jointly optimizing centers and
    radii is also acceptable
- SciPy-based constrained optimization is allowed if it remains reliable enough for
    the timeout budget
- Good initial placements matter
- Edge effects are important in a square container
- Variable-sized circles are likely beneficial
- Focus on valid packings first, then improve sum of radii
- Good solutions should remain reasonable for nearby unseen values of n

Do not use plotting code inside the evolved block.
Do not print debugging output from the evolved block.
Write all improvements only between # EVOLVE-BLOCK-START and # EVOLVE-BLOCK-END.
```

Listing 2. Example system prompt for the EMO-STA signal-processing task family.

```
You are improving a generic causal 1D signal-processing / denoising
algorithm for a multi-task shared-then-specialize workflow.

Important requirements:
- Preserve the function signatures exactly:
  - process_signal(noisy_signal, window_size=20)
  - run_signal_processing(noisy_signal, window_size=20)
- The same evolving program must work across multiple signal families, not
  just one.
- Do not hardcode any single benchmark family, task ID, signal formula, or
  task-specific constants beyond generic windowed processing.
- The evaluator passes the actual noisy signal into the candidate; the
  algorithm must filter that direct input instead of regenerating data.
- Focus on causal smoothing, trend preservation, low lag, bounded memory,
  deterministic behavior, and stable runtime.
- Useful generic ideas include weighted moving averages, Savitzky-Golay
  style local fitting, adaptive smoothing, robust local regression, and
  trend-aware smoothing, but no single method is required.
- Use only Python, NumPy, and SciPy.
```

Listing 3. Example system prompt for the EMO-STA Heilbronn-triangle task family.

```
You are an expert mathematician specializing in combinatorial and continuous geometry.

Your task is to improve a generic constructor/optimizer for the Heilbronn triangle
    problem in a unit-area triangle.

The evolving code must preserve these exact function signatures:

def construct_points(n: int):
    ...
    return points, min_area
```

```
def run_heilbronn(n: int):
    return construct_points(n)

The evaluator uses one fixed canonical unit-area triangle:
A = (0.0, 0.0)
B = (2.0, 0.0)
C = (0.0, 1.0)

A point (x, y) is inside the triangle iff:
- x >= 0
- y >= 0
- x / 2 + y <= 1

The evaluator will call your code for multiple related tasks with:
n in {9, 10, 11, 12}

Public pinned scoring anchors:
- n=9: 0.0548469387755102
- n=10: 0.04337673349889024
- n=11: 0.03609267801015405
- n=12: 0.03100478174352528

Constraints:
- points must have shape (n, 2)
- all points must be finite
- all points must lie on or inside the canonical triangle
- keep the algorithm deterministic

Guidance:
- Favor reusable geometric construction and optimization routines parameterized by n
- Avoid implementing four completely separate hardcoded exact configurations as the
    main strategy
- Boundary-heavy configurations, symmetry, barycentric parameterizations,
    deterministic local improvement, and maximizing the worst triple area are all
    reasonable ideas
- Duplicates or collinear triples are feasible but will produce zero score, so try to
    avoid them
- Because all unit-area triangles are affinely equivalent, it is acceptable to
    optimize only in this canonical triangle
- Focus on valid point sets first, then improve the minimum triangle area

Do not use plotting code inside the evolved block.
Do not print debugging output from the evolved block.
Write all improvements only between # EVOLVE-BLOCK-START and # EVOLVE-BLOCK-END.
```

Listing 4. Example system prompt for the EMO-STA circle-packing rectangle task family.

```
You are an expert mathematician specializing in circle packing and computational
    geometry.

Your task is to improve a generic constructor/optimizer for packing n circles of
    varying radii inside a rectangle of perimeter 4, maximizing the sum of their radii.


The evolving code must preserve these exact function signatures:

def construct_packing(n: int):
    ...
    return centers, radii, alpha, sum_radii

def run_packing(n: int):
    return construct_packing(n)

In this family:
```

```
- alpha is the rectangle width
- rectangle height is 2 - alpha
- alpha must satisfy 0 < alpha <= 1
- the evaluator will call your code for multiple related tasks with:
  n in {20, 21, 22, 23}

Pinned public scoring anchors:
- n=20: 2.305
- n=21: 2.365
- n=22: 2.425
- n=23: 2.484

Constraints:
- centers must have shape (n, 2)
- radii must have shape (n,)
- all circles must lie fully inside the rectangle [0, alpha] x [0, 2 - alpha]
- circles must not overlap
- keep all outputs finite
- keep the algorithm deterministic

Guidance:
- Favor reusable geometric construction and optimization routines parameterized by n
- The rectangle aspect ratio is also part of the search through alpha
- Avoid implementing four completely separate hardcoded exact layouts as the main
    strategy
- Explicit constructive geometry, symmetry-breaking, corner/edge utilization, ring/
    grid hybrids, and local continuous optimization are all reasonable
- Computing radii from fixed centers is acceptable; jointly optimizing centers, radii,
     and alpha is also acceptable
- SciPy-based constrained or unconstrained optimization is allowed if it remains
    reliable enough for the timeout budget
- Good initial placements matter
- The shorter side is alpha, so radii cannot exceed alpha / 2
- Focus on valid packings first, then improve sum of radii

Do not use plotting code inside the evolved block.
Do not print debugging output from the evolved block.
Write all improvements only between # EVOLVE-BLOCK-START and # EVOLVE-BLOCK-END.
```

D.2. Illustrative EMO-STA Program Examples

These examples are intended to illustrate how EMO-STA changes program structure, not to present minimal or globally optimal implementations. We selected circle packing and signal processing because they show two distinct adaptation modes: numerical/geometric refinement and behavioral trade-off adjustment. In both cases, adaptation does not discard the shared solution and start over; it retains the main computational template discovered during shared evolution, then reallocates search effort, modifies heuristics, or tunes decision thresholds for the target task.

Table 7 summarizes the two examples. We compare the program obtained after shared evolution (*STA Best-Shared (Before Adaptation)*) with the final program obtained after *STA Best-Local* adaptation, emphasizing qualitative changes in representation and search behavior in addition to the final score. The single-task score is reported as a reference average over the five independent single-task runs from the same matched-compute setting.

Circle packing. The circle-packing example uses `claude-opus-4-6` with a $60/15/120$ Shared / Adapt / Total budget on the $n = 24$ task, corresponding to a matched single-task baseline budget of $B = 30$ per task. The shared program already captures a useful family-level idea: circle packing should be treated as a geometric optimization problem rather than as only a fixed constructive layout. It proposes plausible geometric arrangements, relaxes center positions, assigns feasible radii through constrained optimization, and then improves the layout through local search and restarts. This shared solution scores 0.9643 on $n = 24$, comparable to the five-run single-task average of 0.9567 ± 0.0194. Best-Local adaptation keeps this same algorithmic representation, but makes the search more specialized for the target circle count. The adapted program adds more diverse layout proposals and spends more of its computation jointly adjusting center positions and radii. The score rises to 0.9995. This is the intended EMO-STA behavior: shared evolution discovers the general solver structure,

Family	Task	Model	Budget	STA Best-Shared (Before Adaptation)	STA Best-Local
Circle packing	$n = 24$	`claude-opus-4-6`	$60/15/120$	0.9643	**0.9995**
Signal processing	Step changes	`claude-opus-4-6`	$60/10/100$	0.6938	**0.8834**

Table 7. Representative program-level examples of EMO-STA adaptation. Budgets are reported as Shared / Adapt / Total iterations, where Total is the matched family-level compute $S + KA = KB$. The corresponding per-task single-task baseline budgets are $B = 30$ for circle packing and $B = 25$ for signal processing. The STA Best-Shared (Before Adaptation) column reports the best shared program evaluated before target-specific adaptation.

while specialization sharpens that solver for one target instance.

Listing 5. Selected code snippets from the circle-packing shared and adapted programs.

```
# Shared program: hexagonal layouts, relaxation, LP radii, local search.

import numpy as np
from scipy.optimize import linprog
from scipy.spatial.distance import pdist, squareform
import time

def compute_max_radii_lp(centers):
   n = len(centers)
   c_obj = -np.ones(n)
   dists = squareform(pdist(centers))

   pairs = []
   for i in range(n):
      for j in range(i + 1, n):
         pairs.append((i, j, dists[i, j]))

   A_ub = np.zeros((len(pairs) + n, n))
   b_ub = np.zeros(len(pairs) + n)

   for idx, (i, j, d) in enumerate(pairs):
      A_ub[idx, i] = 1.0
      A_ub[idx, j] = 1.0
      b_ub[idx] = d - 1e-10

   for i in range(n):
      x, y = centers[i]
      A_ub[len(pairs) + i, i] = 1.0
      b_ub[len(pairs) + i] = max(
         min(x, y, 1.0 - x, 1.0 - y), 0.0
      ) - 1e-10

   res = linprog(c_obj, A_ub=A_ub, b_ub=b_ub,
               bounds=[(0, None)] * n, method="highs")
   if res.success:
      return np.maximum(res.x, 0.0)

   return np.zeros(n)

def gen_hex(n, cols, rows, shift=0.0):
   dx = 1.0 / (cols + 1)
   dy = 1.0 / (rows + 1)
   pts = []
   for r in range(rows):
      off = (0.5 * dx if r % 2 else 0.0) + shift
      for c in range(cols):
         if len(pts) >= n:
            break
         pts.append([
```

```
            np.clip(dx * (c + 1) + off, 0.01, 0.99),
            np.clip(dy * (r + 1), 0.01, 0.99),
         ])
   return np.array(pts[:n]) if len(pts) >= n else None

def construct_packing(n: int):
   n = int(n)
   t0 = time.time()
   best_c, best_s = None, -1.0

   sq = int(np.ceil(np.sqrt(n)))
   for cols in range(max(2, sq - 2), min(n + 1, sq + 6)):
      rows = int(np.ceil(n / cols))
      for shift in [0.0, 0.02, -0.02]:
         centers = gen_hex(n, cols, rows, shift)
         if centers is None:
            continue
         centers = force_relax(centers, n, iters=80)
         score = float(np.sum(compute_max_radii_lp(centers)))
         if score > best_s:
            best_s, best_c = score, centers.copy()

   best_c, best_s = optimize_cyclic(
      best_c, n, t0, 18, steps=[0.04, 0.02, 0.01]
   )

   rng = np.random.RandomState(42)
   for it in range(50):
      if time.time() - t0 > 46:
         break
      scale = max(0.003, 0.09 * (0.78 ** it))
      trial = np.clip(best_c + rng.randn(n, 2) * scale, 0.005, 0.995)
      trial = force_relax(trial, n, iters=40)
      trial, score = optimize_cyclic(
         trial, n, t0, min(time.time() - t0 + 3, 46),
         steps=[0.02, 0.008],
      )
      if score > best_s:
         best_s, best_c = score, trial.copy()

   radii = compute_max_radii_lp(best_c)
   return best_c, radii, float(np.sum(radii))


# Adapted program: same solver structure, plus joint SLSQP refinement
# and more diverse layout proposals.

from scipy.optimize import minimize
from scipy.sparse import csr_matrix

def joint_constraints(x, n):
   centers = x[:2 * n].reshape(n, 2)
   radii = x[2 * n:]
   cons = []

   for i in range(n):
      cons.append(centers[i, 0] - radii[i])
      cons.append(centers[i, 1] - radii[i])
      cons.append(1.0 - centers[i, 0] - radii[i])
      cons.append(1.0 - centers[i, 1] - radii[i])

   for i in range(n):
      for j in range(i + 1, n):
         d = np.linalg.norm(centers[i] - centers[j])
         cons.append(d - radii[i] - radii[j])
```

```
    return np.array(cons)

def joint_optimize(centers, n, maxiter=300):
    radii = compute_max_radii_lp(centers)
    x0 = np.concatenate([centers.flatten(), radii])
    bounds = [(0.005, 0.995)] * (2 * n) + [(1e-4, 0.5)] * n

    res = minimize(
        lambda x, nn: -np.sum(x[2 * nn:]),
        x0,
        args=(n,),
        method="SLSQP",
        bounds=bounds,
        constraints={"type": "ineq", "fun": joint_constraints, "args": (n,)},
        options={"maxiter": maxiter, "ftol": 1e-12},
    )

    if np.all(joint_constraints(res.x, n) >= -1e-6):
        centers_new = res.x[:2 * n].reshape(n, 2)
        radii_new = np.maximum(res.x[2 * n:], 0.0)
        return centers_new, radii_new, float(np.sum(radii_new))

    return centers, radii, float(np.sum(radii))

def gen_sunflower(n):
    golden = (1.0 + np.sqrt(5.0)) / 2.0
    pts = []
    for i in range(n):
        r = np.sqrt((i + 0.5) / n) * 0.45
        theta = 2.0 * np.pi * i / golden**2
        pts.append([
            np.clip(0.5 + r * np.cos(theta), 0.01, 0.99),
            np.clip(0.5 + r * np.sin(theta), 0.01, 0.99),
        ])
    return np.array(pts)

def construct_packing(n: int):
    n = int(n)
    t0 = time.time()
    candidates = []

    sq = int(np.ceil(np.sqrt(n)))
    for cols in range(max(2, sq - 2), min(n + 1, sq + 4)):
        rows = int(np.ceil(n / cols))
        centers = gen_hex_grid(n, cols, rows)
        if centers is not None:
            candidates.append(centers)

    candidates.append(gen_sunflower(n))
    optimized_grid = gen_optimized_grid(n)
    if optimized_grid is not None:
        candidates.append(optimized_grid)

    best_c, best_s = None, -1.0
    for centers in candidates:
        centers = force_relax(centers, n, iters=80)
        score = float(np.sum(compute_max_radii_lp(centers)))
        if score > best_s:
            best_s, best_c = score, centers.copy()

    best_c, best_r, best_s = joint_optimize(best_c, n, maxiter=500)
    best_c, best_s = optimize_cyclic(
        best_c, n, t0, 15, steps=[0.04, 0.02, 0.01]
    )
```

```
    best_c, best_r, best_s = joint_optimize(best_c, n, maxiter=800)

    if time.time() - t0 < 58:
        best_c, best_s = optimize_cyclic(
            best_c, n, t0, 59, steps=[0.0003, 0.0001]
        )

    radii = compute_max_radii_lp(best_c)
    return best_c, radii, float(np.sum(radii))
```

Signal processing. The signal-processing example uses `claude-opus-4-6` with a $60/10/100$ Shared / Adapt / Total budget on the step-change task. The shared program learns a general causal filtering strategy: estimate local structure from a sliding window, combine a trend-following estimate with a smoother estimate, and then apply adaptive smoothing over time. This kind of program is useful across the signal-processing family because the tasks contain different mixtures of noise, oscillation, trend, and abrupt change. The shared program scores 0.6938 on the step-change task. Best-Local adaptation keeps the same multi-stage causal-filtering structure, but changes the behavior of the filter for the target task. In particular, it becomes more conservative about reacting to small fluctuations while still allowing larger changes to pass through. This raises the score to 0.8834, above the five-run single-task average of 0.7373 ± 0.0653. This example shows a different kind of specialization from circle packing: rather than refining a geometric optimizer, adaptation refines the behavioral tradeoff encoded in the algorithm.

Listing 6. Selected code snippets from the signal-processing shared and adapted programs.

```
# Shared program: local trend estimation plus adaptive smoothing.

import numpy as np
from numpy.lib.stride_tricks import sliding_window_view

def process_signal(noisy_signal, window_size):
    signal = np.asarray(noisy_signal, dtype=float)
    w = max(3, int(window_size))
    n = len(signal)

    if n < w:
        return signal.copy()

    t = np.arange(w, dtype=float)
    t_end = float(w - 1)
    windows = sliding_window_view(signal, w)

    sigma = w * 0.45
    weights = np.exp(-0.5 * ((t - t_end) / sigma) ** 2)

    X_quad = np.column_stack([np.ones(w), t, t * t])
    WX_quad = X_quad * weights[:, None]
    kernel_quad = np.array([1.0, t_end, t_end * t_end]) @ np.linalg.solve(
        WX_quad.T @ X_quad + 1e-10 * np.eye(3),
        WX_quad.T,
    )

    X_lin = np.column_stack([np.ones(w), t])
    WX_lin = X_lin * weights[:, None]
    kernel_lin = np.array([1.0, t_end]) @ np.linalg.solve(
        WX_lin.T @ X_lin + 1e-10 * np.eye(2),
        WX_lin.T,
    )

    local_quad = windows @ kernel_quad
    local_lin = windows @ kernel_lin

    noise_est = np.median(np.abs(np.diff(signal))) * 1.4826 + 1e-12
    curvature = np.abs(local_quad - local_lin)
    blend = np.clip(curvature / (3.0 * noise_est), 0.0, 1.0)
```

```
    filtered = blend * local_quad + (1.0 - blend) * local_lin

    for alpha_base, threshold, sharpness in [
        (0.03, 1.2, 5.0),
        (0.04, 0.5, 6.0),
        (0.06, 0.35, 7.0),
    ]:
        smoothed = np.empty_like(filtered)
        smoothed[0] = filtered[0]
        for i in range(1, len(filtered)):
            g = abs(filtered[i] - smoothed[i - 1]) / noise_est
            alpha = alpha_base + (1.0 - alpha_base) / (
                1.0 + np.exp(-sharpness * (g - threshold))
            )
            smoothed[i] = alpha * filtered[i] + (1.0 - alpha) * smoothed[i - 1]
        filtered = smoothed

    return filtered


# Adapted program: same filtering template, but more conservative around
# small fluctuations and more selective about when to react.

def process_signal(noisy_signal, window_size):
    signal = np.asarray(noisy_signal, dtype=float)
    w = max(3, int(window_size))
    n = len(signal)

    if n < w:
        return signal.copy()

    t = np.arange(w, dtype=float)
    t_end = float(w - 1)
    windows = sliding_window_view(signal, w)

    noise_est = (
        np.median(np.abs(np.diff(signal))) * 1.4826 / np.sqrt(2.0) + 1e-12
    )

    sigma = w * 0.4
    weights = np.exp(-0.5 * ((t - t_end) / sigma) ** 2)

    X_lin = np.column_stack([np.ones(w), t])
    WX_lin = X_lin * weights[:, None]
    kernel_lin = np.array([1.0, t_end]) @ np.linalg.solve(
        WX_lin.T @ X_lin + 1e-10 * np.eye(2),
        WX_lin.T,
    )

    local_lin = windows @ kernel_lin
    local_mean = windows @ (weights / weights.sum())

    slope_mag = np.abs(local_lin - local_mean)
    blend = np.clip(slope_mag / (2.0 * noise_est), 0.0, 1.0)
    filtered = blend * local_lin + (1.0 - blend) * local_mean

    dead_zone = 0.3 * noise_est

    for alpha_base, threshold, sharpness, zone_scale in [
        (0.02, 1.5, 6.0, 1.0),
        (0.02, 0.8, 8.0, 0.5),
        (0.03, 0.5, 10.0, 0.3),
    ]:
        smoothed = np.empty_like(filtered)
```

```
        smoothed[0] = filtered[0]

        for i in range(1, len(filtered)):
            diff = filtered[i] - smoothed[i - 1]
            g = abs(diff) / noise_est
            alpha = alpha_base + (1.0 - alpha_base) / (
                1.0 + np.exp(-sharpness * (g - threshold))
            )

            if abs(diff) < dead_zone * zone_scale:
                alpha = min(alpha, alpha_base * 0.5)

            smoothed[i] = alpha * filtered[i] + (1.0 - alpha) * smoothed[i - 1]

        filtered = smoothed

    return filtered
```