HEUREVO: AGENTIC EVOLUTION OF HYBRID SOLVER-AUGMENTED HEURISTICS FOR TIME-CRITICAL MATHEMATICAL OPTIMIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances in agentic heuristic design use AI agents and execution feedback to automate algorithm discovery for challenging optimization problems. In many practical settings, high-quality solutions must be obtained under strict runtime constraints, motivating hybrid approaches that combine problem-specific heuristics with powerful mathematical programming solvers. However, existing approaches typically improve heuristic components within predefined procedures or tune solver configurations in isolation. This limits holistic adaptation of where to allocate computation, how to leverage solvers, and how to refine the overall algorithmic structure. To address these limitations, we propose HeurEvo, an *automated plan–code–component co-evolution framework* that jointly evolves the high-level algorithmic structures, their implementations, and a shared pool of reusable components. A planner determines which algorithmic components to use, how to combine them, and how to allocate runtime across stages, a coder realizes the resulting plan as executable code, while a component evolver updates the shared component pool. Within an island-based evolutionary framework, plans and implementations co-evolve with feedback from an interpreter agent that analyzes execution results and identifies opportunities for improvement. Across diverse combinatorial optimization benchmarks, challenging MIPLIB instances, HeurEvo finds high-quality solutions within tight runtime budgets, often matching or surpassing state-of-the-art optimization solvers given hours or days of computation. On several nonlinear geometry problems such as hexagon packing, it also improves upon the best previously reported results. These results highlight the value of jointly searching over algorithmic structure and implementation for agentic heuristic design.

1 INTRODUCTION

Mathematical optimization supports decisions in production scheduling, delivery routing, and resource allocation, where a solution is useful only if it can be obtained in time. Under a short runtime budget, a practical approach is to combine problem-specific heuristics and mathematical programming solvers in a multi-step algorithm (Talbi, 2002; Fischetti & Lodi, 2003). Heuristics can quickly construct or repair solutions, while a solver can optimize selected subproblems or polish a promising incumbent within the broader search (Danna et al., 2005). Such hybrid algorithms have proved useful in practical applications, such as industrial production planning (Lee et al., 2023). Their success, however, depends on designing a complete solver-augmented algorithm suited to both the problem and its available solving time.

Recent methods use AI agents to automate heuristic and algorithm discovery, as well as solver configuration tuning. However, prior automatic heuristic-design methods typically restrict their search to individual components, such as scoring or update rules, within a fixed algorithmic framework such as ant colony optimization (ACO) or guided local search (GLS) (Liu et al., 2024; Ye et al., 2024; Wu et al., 2026). Solver-oriented work, in turn, searches over solver configurations while retaining the solver's overall procedure (Luo et al., 2026). General-purpose code-evolution methods (Cemri et al., 2026; Liu et al., 2026; Sharma, 2025) broaden the search space by modifying entire programs, but they do not explicitly separate high-level algorithm structure from low-level implementation.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

This leaves the design of complete hybrid optimization algorithms underexplored, especially how to combine heuristic and solver methods and allocate a shared runtime budget among them, which motivates our research question:

How can agents jointly evolve the composition and implementation of hybrid optimization algorithms to obtain better solutions within a fixed execution-time budget?

We answer this question with HeurEvo, an automated *plan–code–component co-evolution framework* that jointly evolves high-level algorithmic plans, their executable implementations, and a shared pool of reusable components. A planner constructs and revises plans by selecting components, ordering their subgoals, and allocating runtime across stages. A coder implements each plan and refines its code and runtime allocation, while the planner revisits the algorithmic composition when code-level improvements stall. A component evolver uses experience accumulated during search to refine the shared component pool, allowing subsequent plans to build on improved components. An interpreter analyzes execution results and step traces to guide this evolution. Within an island-based evolutionary framework, different plans and their implementations evolve in separate populations while sharing knowledge through the component pool.

Figure 1 illustrates the resulting behavior. The evolved hybrid algorithm first combines greedy construction and large neighborhood search (LNS) to obtain a promising solution, then uses Gurobi to optimize selected subproblems, followed by solver-based polishing and local cleanup. By jointly choosing these components, their ordering, and their runtime allocation, HeurEvo makes effective use of the available execution budget and achieves a lower objective gap than the illustrated programs produced by whole-program evolution and ReEvo's fixed ACO framework with a single LLM-evolved subcomponent, which is the heuristic function guiding node selection (Ye et al., 2024).

Our goal is to discover fast hybrid heuristics that generalize across related problem instances, producing high-quality solutions within a tight execution budget (two minutes in our experiments) without rerunning the discovery process for each new instance. We evaluate HeurEvo on 11 synthetic tasks, six MIPLIB-NL-derived problems (Li et al., 2026), and four nonlinear geometry families. On the synthetic tasks, HeurEvo achieves mean gaps of -2.22% on training instances and -1.45% on held-out test instances relative to Gurobi's 48-hour incumbents (Gurobi Optimization, LLC, 2026), outperforming these incumbents on eight training tasks and seven test tasks. On the MIPLIB-NL problems, HeurEvo improves upon AdaEvolve on four of six training problems. For nonlinear geometry, combining a HeurEvo phase with subsequent AdaEvolve refinement produces several objectives numerically better than the reported reference values. Ablation studies further support the benefit of combining adaptive plan revision with component evolution.

Our contributions.

- **Structured search over complete hybrid algorithms.** We formulate automated heuristic design as an explicit search over complete solver-augmented algorithms. An evolvable plan selects heuristic and solver components, specifies their execution order and subgoals, and allocates a shared runtime budget across them. This makes algorithm composition a first-class search variable.
- **Plan–code–component co-evolution.** We introduce HeurEvo, a plan–code–component co-evolution framework that jointly evolves algorithmic plans, their executable implementations, and a shared pool of reusable components. Code evolution improves implementations under the current plan; adaptive plan review revises the algorithmic structure when code-level progress stalls; and component evolution refines reusable components using execution experience accumulated across islands. Together, these mechanisms allow experience from previous executions to improve both current algorithms and future planning.
- **Strong empirical performance under tight runtime budgets.** Across synthetic combinatorial optimization, MIPLIB-NL-derived, and nonlinear geometry benchmarks, HeurEvo achieves strong performance with a two-minute execution budget, often matching or improving upon far longer-running solver baselines and prior code-evolution methods. It also contributes to new best reported results on several nonlinear geometry problems, with ablations supporting the value of adaptive plan review and component evolution.

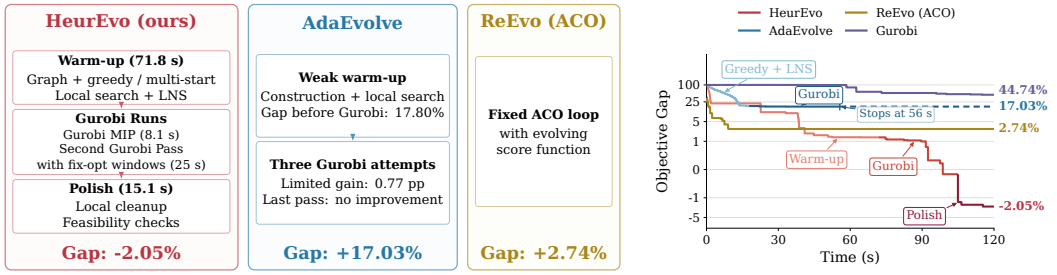


Figure 1: Comparison of heuristics generated from HeurEvo, AdaEvolve, and ReEvo (ACO) for an orienteering problem. **Left:** Stages (or steps) of the selected heuristic programs; bold values at the bottom report their measured objective gaps (lower is better). **Right:** Objective-gap trajectories on the same orienteering instance. AdaEvolve's program remains inactive after 52.2 seconds.

2 Related Work

Island-based evolutionary program search. LLM-guided evolutionary program search has emerged as a general paradigm for automated algorithm discovery, from seminal systems such as FunSearch (Romera-Paredes et al., 2024) and AlphaEvolve (Novikov et al., 2025) to a growing set of follow-up frameworks (Lehman et al., 2023; Lange et al., 2026; Jiang et al., 2026; Wang et al., 2025; Assumpção et al., 2025; Khrulkov et al., 2025; Wan et al., 2025). At their core, these frameworks use LLMs to generate candidate programs and execution feedback to iteratively improve them through population-based search. Island-based variants extend this process across multiple semi-independent populations, preserving diverse search trajectories and reducing premature convergence. Notably, OpenEvolve (Sharma, 2025) provides a competitive open-source implementation that combines MAP-Elites (Mouret & Clune, 2015), exploratory and elite parent sampling, and migration across islands. AdaEvolve (Cemri et al., 2026) and EvoX (Liu et al., 2026) further extend this paradigm: AdaEvolve adaptively controls exploration and evaluation allocation while introducing high-level tactics to redirect stalled search, whereas EvoX co-evolves candidate solutions with search policies that determine which candidates to build on and how new variants are generated.

Our work differs primarily in the object being evolved. Rather than treating the candidate primarily as a program, HeurEvo explicitly represents a hybrid optimization algorithm as a composition of heuristic and solver components together with an executable implementation. It then co-evolves the components, their composition into a plan, and the corresponding code, allowing execution feedback to revise both algorithmic structure and implementation.

Automatic Heuristic Design for Optimization Problems. A growing body of work applies LLM-based automatic heuristic design to optimization problems (Liu et al., 2023; Stein & Bäck, 2025; Dat et al., 2025; Yao et al., 2025; Yang et al., 2025; Kuang et al., 2026). Early approaches such as FunSearch (Romera-Paredes et al., 2024) and EoH (Liu et al., 2024) demonstrated that LLMs can iteratively discover effective optimization heuristics through program generation and execution feedback. ReEvo (Ye et al., 2024) augments this process with short- and long-term reflection, MCTS-AHD (Zheng et al., 2025) uses Monte Carlo tree search to explore promising heuristic lineages, and RefineEvo (Wu et al., 2026) adaptively selects and refines evolutionary operators based on search progress. More recent work broadens the unit of evolution beyond a single heuristic component. MILP-Evo (Nie et al., 2026) jointly evolves cut-selection and branching rules within a branch-and-cut solver. MOTIF (Kiet et al., 2026) jointly optimizes multiple interacting algorithmic components, while MuEvo (Lv et al., 2026) co-evolves heuristic ensembles and adaptively manages their constituent components. These methods substantially expand the search space, but operate within predefined algorithmic roles or solver structures. At the other extreme, concurrent work such as ATLAS (Yazdani et al., 2026) evolves entire executable programs, allowing broad structural changes but leaving algorithm composition implicit in the code.

HeurEvo instead makes the structure of a complete hybrid optimization algorithm an explicit object of evolution. Its plan can select heterogeneous heuristic and solver components, organize them into ordered steps with explicit subgoals, and allocate a shared runtime budget across them, while code

evolution separately refines how each step is implemented. Component evolution further updates the reusable building blocks from which future plans are constructed.

3 METHOD

3.1 PROBLEM FORMULATION AND HYBRID ALGORITHM COMPOSITION

Problem formulation. Let $\mathcal{P}$ denote a distribution over instances of a target optimization problem family. We construct a training set $\mathcal{D} = \{d_j\}_{j=1}^{N}$ of N instances, where $d_j \sim \mathcal{P}$.

Given $\mathcal{D}$ and a per-instance runtime budget B, our objective is to discover an executable program that achieves high average solution quality. We represent a hybrid algorithm G as an ordered composition of m heuristic or solver operations, i.e. $G = g_1 \odot g_2 \odot \cdots \odot g_m$, where $\odot$ denotes sequential composition and intermediate results are passed between steps. Let $T_i(G; d)$ denote the runtime of each operation g_i when executed as part of G on instance d, including its associated overhead, and let B denote the total runtime budget. Furthermore, let $f(G; d)$ denote the fitness score of the returned solution, with larger values indicating better solution quality. Under a fixed execution environment, we formulate the program search problem as

$$\max_{G} \quad F(G) = \frac{1}{|\mathcal{D}|} \sum_{d \in \mathcal{D}} f(G; d) \qquad \text{s.t.} \qquad T(G; d) = \sum_{i=1}^{m} T_i(G; d) \leq B, \ \ \forall d \in \mathcal{D}. \tag{1}$$

Component, plan, and code. A *component* is a reusable high-level optimization method that admits multiple concrete implementations, such as greedy construction, local search, or calls to a mathematical programming solver. The *component library* contains a set of such reusable components. A *plan* specifies how these components are composed into an algorithm. It selects and orders a sequence of components and, for each component, specifies (1) a *subgoal*, i.e., the intermediate outcome the component should achieve; (2) implementation guidance, such as the neighborhood used for local search; and (3) a time budget. We refer to a component together with these specifications as a *step*. Since steps are executed sequentially, we index them from 1 to the total number of steps.

Executable *code* instantiates the plan as an algorithm G, passing intermediate results between steps while respecting the total per-instance runtime budget B. Code evolution may modify implementation guidance and per-step time allocations while preserving the *plan structure*, defined as the ordered sequence of components and their associated subgoals. Changes to this structure require plan evolution. Table 1 shows an example plan produced during an actual traveling salesperson problem (TSP) run.

Table 1: An example plan for the traveling salesperson problem.

Step	**Component**	Subgoal	*Implementation details*	Time budget
1	**Euclidean tour construction**	Produce feasible and diverse incumbents	*Build a small portfolio of nearest-neighbor and insertion tours; retain a diverse elite set*	9 s
2	**Local tour improvement**	Rapidly reduce tour length with cheap moves	*Apply candidate-restricted 2-opt and Or-opt descent; perturb and restart only after stagnation*	20 s
3	**Candidate-edge restricted MIP**	Escape local minima through nonlocal edge changes	*Warm-start Gurobi on a sparse graph containing proximity, incumbent, elite, and repair arcs*	32 s
4	**Large-neighborhood search**	Repair costly regions missed by the sparse global search	*Fix most of the incumbent, destroy a long-edge or geographic region, and use Gurobi to reconnect that region exactly*	42 s
5	**Residual anytime primal portfolio**	Use the residual budget on the most productive search mode	*Adaptively select among Gurobi LNS micro-repair, Gurobi sparse-MIP retry, and heuristic local-search restart using same-run progress*	Remaining time

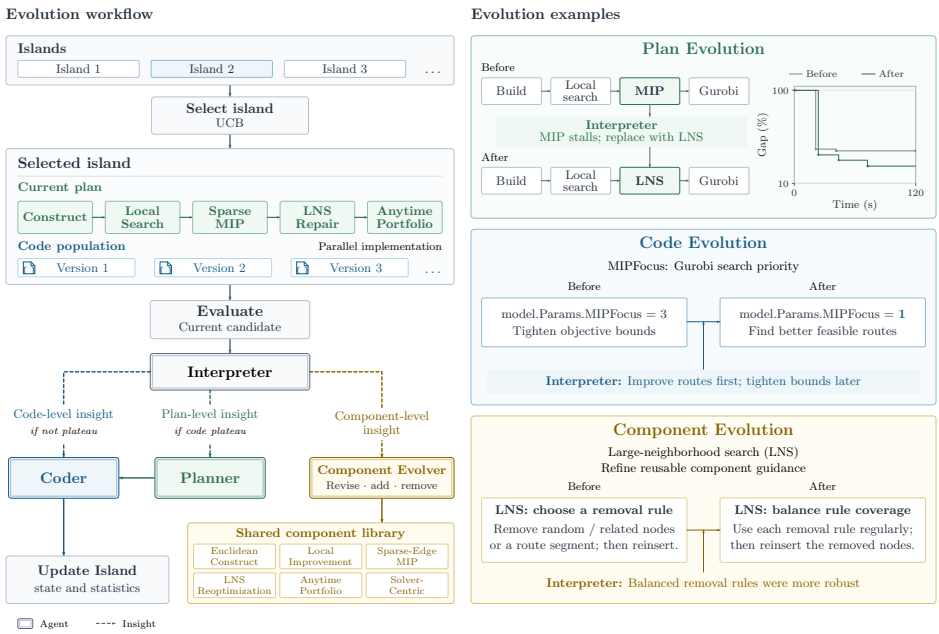


Figure 2: HeurEvo workflow and examples of its three levels of evolution. **Left:** Simplified workflow. UCB selects an island for code evaluation; Interpreter feedback guides code refinement, plan revision when progress stalls, and updates to the shared component library. **Right:** Example edits. *Top:* Plan evolution replaces a stalled mixed-integer programming (MIP) step with large-neighborhood search (LNS). *Middle:* Code evolution changes Gurobi's `MIPFocus` setting while preserving the plan structure. *Bottom:* Component evolution refines reusable LNS guidance to encourage balanced use of removal rules.

3.2 Our Plan-Code-Component Co-Evolution Framework

The representation above makes explicit three coupled design decisions: which components to use for a given problem, how to compose them, and how to implement each step under a shared runtime budget. To address these coupled decisions, we introduce HeurEvo, a *plan–code–component co-evolution framework* that jointly evolves the components themselves, the plans that compose them, and the code that implements those plans.

To structure this search, we initialize a shared *component library* and generate alternative plans from it. Each plan defines a plan-conditioned *island* whose programs share the same ordered subgoals and selected components. Islands maintain separate active code populations and plan-specific feedback while sharing the component library and an archive of evaluated programs. Figure 2 (left) summarizes the resulting evolution workflow.

The examples in Figure 2 (right) illustrate three complementary levels of evolution:

- **Component-level evolution** aims to refine the high-level search space. Based on accumulated feedback and evaluation evidence, the *component evolver* revises existing component descriptions, introduces previously absent components, and removes components that remain ineffective across evaluated plans and implementations.
- **Plan-level evolution** aims to discover more effective compositions of heuristics and solver operations when code-level improvements plateau. It may add, remove, replace, or reorder steps, or change their subgoals or selected components. The *planner* considers programs under the current plan, the performance of plans on other islands, and feedback accumulated from code-level evolution on the current island. It proposes a revised plan with preliminary implementation procedures, which the Plan-based coding agent (coder) translates into a new executable code.
- **Code-level evolution** aims to improve solution quality through better step implementations under a fixed plan. Given a parent code, inspiration programs, and feedback from previous evaluations,

the *coder* generates a child while preserving the plan's ordered steps. It may revise concrete algorithms, data structures, parameters, solver settings, stopping rules, time allocations, and interactions between adjacent steps.

Detailed agent specifications are provided in Appendix C.2; the prompt templates for initial code generation and plan-aware code evolution are given in Appendices D.3 and D.4, respectively.

Evaluation and interpreter feedback. Each child is evaluated on $\mathcal{D}$ under budget B and checked for feasibility and execution failures. Based on these evaluation results, the Interpreter compares each validated child with its sampled parent in terms of objective quality, feasibility, runtime, material code changes, and step-level execution traces. Code-level feedback guides subsequent code updates on the selected island; plan-level feedback accumulates for the island's next plan review; and component-level evidence can be aggregated across islands. Separately, scheduling progress is measured against the island's best current-plan code rather than the sampled parent. Before the next update, the selected island's population, feedback, and scheduling statistics are updated. Retained programs and their reports enter the shared archive, whereas the active plans and populations of other islands remain unchanged. Appendix C.2 provides further details about the Interpreter, while Section 3.3 describes how these updates are scheduled.

3.3 Island-Based Evolution Procedure

HeurEvo maintains K islands, each associated with a current plan and an active population of validated programs. Following AdaEvolve (Cemri et al., 2026), we use an upper confidence bound (UCB) rule to allocate search across islands. Our controller additionally determines whether the selected island should continue improving code under its current plan or revise the plan itself. At initialization, each island is assigned a compositional plan together with a valid seed code (see Appendix C.3).

Island selection. Before each round, let $\bar{r}_k$ denote the discounted average reward for island k and n_k its cumulative number of completed code updates. Let $n_{\text{sum}} = \sum_{k=1}^{K} n_k$ be the total count across islands. We select the next island as $k^\star = \arg\max_{k\in 1,\dots,K} \left[\bar{r}_k + C\sqrt{\frac{\log n_{\text{sum}}}{n_k}}\right]$, where the first term favors exploitation of islands that have recently earned higher rewards, while the second promotes exploration of islands that have been visited less often; C controls the balance between the two. Initialization ensures $n_k > 0$. Our reward measures how much each code update helps an island close the gap to the current global best. Any update that improves the island's best solution receives a positive reward, with larger rewards for improvements that close a larger fraction of its gap to the current global best. Non-improving updates receive zero reward. Exact reward and discounting definitions are provided in. Appendix C.4.

Updating an island. We next describe the three mechanisms that govern evolution within and across islands: code refinement, plan review, and component evolution. The first two determine how the selected island is updated, while component evolution uses accumulated experience to refine the shared component pool.

After selecting an island, the controller decides whether to continue code-level improvement under the current plan or to review the plan itself. We measure recent progress by tracking whether successive code updates improve the island's current best code. If progress exceeds a plateau threshold τ, the controller continues *code refinement*; otherwise, it triggers a *plan review*. The precise progress metric, smoothing rule, and reset behavior are described in Appendix C.4.

Code refinement. When the plan composition remains unchanged, evolution proceeds similarly to AdaEvolve (Cemri et al., 2026). Parents are drawn from the selected island's current population, while programs from other islands or earlier generations may serve as inspiration. The coder refines the implementation while preserving the plan's selected components and ordered subgoals.

Plan review. When progress stalls, HeurEvo revisits the algorithmic composition. The planner uses accumulated feedback from the selected island, together with evidence from other islands, to propose a revised plan. Once validated, the revised plan initializes a new population for that island,

followed by $W \geq 1$ local code updates before global island selection resumes. Programs from the previous plan remain available as inspiration but are no longer eligible as parents.

Component evolution. Experience accumulated within the islands is also used to refine the shared component pool, allowing improved component descriptions and execution feedback to transfer across islands. These updates do not directly modify an island's current plan, which changes only through plan review. Algorithm C2 in Appendix C.4 presents the full evolution procedure.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Dataset. We consider three families of data: **1) Standard synthetic problems.** The 11 tasks are traveling salesman (TSP), clustered and uniform capacitated vehicle routing (C-CVRP and U-CVRP), orienteering (OP), job-shop scheduling (JSS), permutation flow-shop scheduling (PFSS), resource-constrained project scheduling (RCPS), offline bin packing (OBP), Max-Cut, maximum independent set (MIS), and decoupling-capacitor placement (Decap). Instances use literature-based generation settings with task-specific adaptations; details are in Appendix B.2. **2) MIPLIB-derived problems.** Six complex real-world optimization problems are selected from MIPLIB-NL (Li et al., 2026): course scheduling (`comp12-2idx`, `comp21-2idx`), graph drawing (`graphdraw-grafo2`, `graphdraw-mainerd`), mining-project selection (`opm2-z12-s8`), and PCB assembly-line configuration (`sct32`). All six were selected because Gurobi did not certify optimality within two hours in the initial screening. Appendix B.3 provides details. **3) Nonlinear geometry problems.** We further evolve heuristics for four problems studied in AlphaEvolve (Novikov et al., 2025): circle packing in a square, circle packing in a rectangle, min-max distance ratio, and hexagon packing (Appendix B.4).

Baselines. We compare against seven LLM-based methods from two groups. The first comprises general-purpose program-evolution frameworks, including OpenEvolve (Sharma, 2025), AdaEvolve (Cemri et al., 2026), and EvoX (Liu et al., 2026). The second comprises automatic heuristic design (AHD) methods, including EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), RefineEvo (Wu et al., 2026), and MCTS-AHD (Zheng et al., 2025). We instantiate the AHD baselines within problem-specific heuristic frameworks or solvers, including ant colony optimization (ACO) (Dorigo et al., 2006), guided local search (GLS) (Arnold & Sörensen, 2019), constructive heuristics, and Gurobi (Gurobi Optimization, LLC, 2026).

Evaluation Metrics. All evolution runs, including baselines, use a budget of 200 generated programs and a two-minute evaluation limit per instance. For method objective f and Gurobi's 48-hour incumbent $f_{48\mathrm{h}}$, the signed gap is $g = s \cdot (f - f_{48\mathrm{h}})/|f_{48\mathrm{h}}|$, with $s = 1$ for minimization and $s = -1$ for maximization. A positive gap indicates a worse objective than Gurobi's incumbent, while a negative gap indicates a better objective. For synthetic problems, evolution uses the mean signed gap across five training instances sampled with different data seeds from the same task distribution; two validation instances select the program and four held-out test instances evaluate it. MIPLIB uses the same number of train/val/test instances. For nonlinear problems, each heuristic evolves on an individual instance and is compared with the best objective reported in prior literature. We use GPT-5.3-Codex for standard synthetic problems and GPT-5.6-sol for the more challenging MIPLIB-derived and nonlinear geometry problems: Implementation details are in Appendix A.

4.2 MAIN RESULTS

Performance on Synthetic Dataset. Table 2 summarizes the mean objective gaps across the 11 synthetic problems, comparing HeurEvo with the code-evolution baselines OpenEvolve, AdaEvolve, and EvoX. HeurEvo achieves mean training and test gaps of -2.22% and -1.45%, respectively; baseline training means range from -0.15% to $+1.22\%$, and their reported test means are all positive. It discovers heuristics that produce better solutions than Gurobi does within 48 hours on 8 of the 11 training problem classes. When transferred to the test set, these heuristics remain effective, outperforming Gurobi's 48-hour solutions on 7 of the 11 problem classes. As shown in Figure 3, HeurEvo makes substantial gains early in evolution and maintains the lowest mean training gap

throughout the subsequent search, indicating more efficient use of the program-generation budget. Table 3 further examines five representative problem classes spanning routing, scheduling, and graph optimization. HeurEvo attains the best or tied-best training gaps on all five, with clear test-time advantages on OP, C-CVRP, and RCPS, showing that these gains extend beyond the training instances. Full results, evaluation coverage, and problem-selection details are provided in Appendix A.1.

Table 2: Average synthetic performance.

	Mean gap (%)		Beat Gurobi	
Method	Train	Test	Train	Test
OpenEvolve	$+0.71$	$+0.41^{\ddagger}$	$5/11$	$4/11$
AdaEvolve	-0.15	$+0.91$	$7/11$	$\mathbf{7/11}$
EvoX	$+1.22$	$+2.24$	$5/11$	$4/11$
HeurEvo	$\mathbf{-2.22}$	$\mathbf{-1.45}$	$\mathbf{8/11}$	$\mathbf{7/11}$

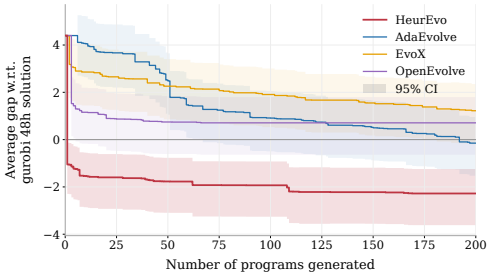


Figure 3: Training gap versus program budget.

Table 3: Representative synthetic objective gaps (%). Since AHD methods do not introduce a host function or an initial heuristic solution for the selected problems, their results are left blank.

Method	OP			C-CVRP			RCPS			PFSS			MIS		
	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test
EoH (GLS)	–	–	–	–	–	–	–	–	–	**-2.65**	**-2.61**	-2.20	–	–	–
ReEvo (ACO)	+4.16	+1.32	+7.40	+6.58	+9.77	+8.28	–	–	–	–	–	–	–	–	–
RefineEvo (ACO)	–	–	–	+5.06	+9.59	+8.18	–	–	–	–	–	–	–	–	–
OpenEvolve	$+16.65$	$+16.10$	$+22.61^{\ddagger}$	$+1.93$	$+4.57$	$+2.96$	$+0.64$	$\mathbf{-0.57}$	$+1.74$	-0.48	$+0.41$	-0.27	$\mathbf{-0.04}$	$+0.39$	$\mathbf{+0.14}$
AdaEvolve	$+16.92$	$+15.73$	$+21.75$	-1.77	-0.84	-1.62	$+1.49$	$+1.21$	$+0.70$	-2.19	-1.84	$\mathbf{-2.21}$	$+0.15$	$+0.29$	$+0.45$
EvoX	$+17.96$	$+16.94$	$+21.57$	$+4.50$	$+2.14$	$+2.17$	$+2.70$	$+3.05$	$+0.91$	-0.30	-0.15	-0.71	$+0.47$	$+1.17$	$+1.88$
HeurEvo	$\mathbf{-3.80}$	$\mathbf{-7.50}$	$\mathbf{-1.94}$	$\mathbf{-3.36}$	$\mathbf{-2.81}$	$\mathbf{-4.38}$	$\mathbf{+0.12}$	$\mathbf{-0.57}$	$\mathbf{0.00}$	-2.25	-2.31	-1.85	$\mathbf{-0.04}$	$\mathbf{0.00}$	$\mathbf{+0.14}$

Performance on MIPLIB Problems. Table 4 extends the comparison to complex real-world MIPLIB problems using GPT-5.6-Sol. HeurEvo achieves the best average training objective on three of the six problems and remains within 1% of AdaEvolve on two others. Notably, HeurEvo reduces the objective relative to AdaEvolve by 0.95% on `comp12-2idx` and 10.93% on `graphdraw-mainerd`, while improving the objective on `sct32` from -0.76 to -1.43, an absolute reduction of 0.67. Admittedly, AdaEvolve performs better on the remaining problems, although the relative shortfalls of HeurEvo are small on `comp21-2idx` (0.39%) and `opm2-z12-s8` (0.62%); the largest shortfall is 9.59% on `graphdraw-grafo2`. Appendix A.2 reports the corresponding training, validation, and test objectives, together with evaluation coverage.

Table 4: Training objectives on MIPLIB.

Method	`comp12-2idx` ↓	`comp21-2idx` ↓	`graphdraw-grafo2` ↓	`graphdraw-mainerd` ↓	`opm2-z12-s8` ↑	`sct32` ↓
AdaEvolve	366.20	**100.40**	85061.30	47940.40	58057.20	**-3.01**
HeurEvo	**327.00**	103.20	**82003.10**	**45681.00**	**58519.40**	-2.77

Performance on Nonlinear Problems. As shown in Table 5, HeurEvo outperforms AdaEvolve in all eight nonlinear task-size configurations, achieving objectives numerically better than the reported state-of-the-art values across circle packing in a rectangle, min-max distance ratio, and hexagon packing. For the min-max distance ratio, HeurEvo yields smaller but consistent improvements across all three configurations. These results highlight both improvements over the evolutionary baseline and fine-grained numerical refinements of strong existing solutions. Complete results across additional configurations, together with the sources of the state-of-the-art values, are provided in Table A11. In view that several reference objectives are rounded, comparisons against them are interpreted at their reported precision.

Table 5: Nonlinear objectives versus reported best values. Underline indicates the part better than AdaEvolve; **Bold** indicates the part better than the reported best-known value.

Method	Circle packing-rectangle ↑		Min-max distance ratio ↓			Hexagon packing ↓		
	$n = 21$	$n = 26$	$n = 16, d = 2$	$n = 21, d = 2$	$n = 22, d = 2$	$n = 11$	$n = 12$	$n = 14$
SOTA	2.3658323759	2.6393205590	12.8892299	17.77499	19.05398	3.9245008973	3.9416420675	4.2689949573
AdaEvolve	2.3642481267	2.6355313151	12.8892299077	17.7749805542	19.0539845098	3.9326758559	3.9416423004	4.2723920772
HeurEvo	2.3658323759	2.6393205**643**	12.889229**1072**	17.7749**803417**	19.0539**769136**	3.9245011309	3.941642062**5**	4.2689950806

Table 6: Component ablation of HeurEvo.

Variant	Averaged performance			OP			C-CVRP			RCPS			PFSS		
	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test	Train	Val	Test
Fixed plan	−1.19	+0.06	+0.25	+1.94	+8.1	+7.97	−3.15	−2.3	−3.26	+1.39	+1.8	+0.39	−0.83	−1.0	−0.50
Component only	−1.45	−0.38	−0.07	−1.67	+0.9	+3.86	−3.09	−2.4	−3.47	+1.37	+1.2	+1.61	−0.79	−0.3	−0.69
Always-plan	−1.67	−0.36	−0.27	+0.45	+8.0	+8.12	−2.84	−2.0	−2.95	+0.50	0.0	+0.43	−1.52	−1.6	−1.67
Always-plan + component	−1.89	−1.38	−0.70	−2.67	**−8.9**	**−2.39**	−3.20	−2.6	−3.62	**−0.35**	0.0	+0.43	−1.16	−0.9	−0.93
Adaptive plan only	−1.35	−1.01	−0.21	−0.87	−3.8	+3.76	−2.48	−2.2	−3.00	+0.87	+0.1	+0.65	−1.48	−1.0	−0.64
HeurEvo (Ours)	**−2.20**	**−2.03**	**−1.45**	**−3.80**	−7.5	−1.94	**−3.36**	**−2.8**	**−4.38**	+0.12	**−0.6**	**0.00**	**−2.25**	**−2.3**	**−1.85**

4.3 Ablation Studies

Effectiveness of Each Module. To assess the contribution of each evolutionary component in HeurEvo, we compare the following five ablation variants: 1) *Fixed plan:* no plan or component evolution; 2) *Component only:* evolve components while keeping plans fixed; 3) *Always-plan:* review plans every round with fixed components; 4) *Always-plan + component:* combine per-round plan reviews with component evolution; 5) *Adaptive plan only:* review plans when progress stalls, with fixed components. Always-plan variants review the selected island's plan every round; adaptive review occurs only when its smoothed relative improvement falls below the plateau threshold, otherwise continuing code refinement. The full pipeline, HeurEvo (Ours), combines adaptive plan review with component evolution. As shown in Table 6, the full pipeline achieves the best mean training, validation, and test gaps of -2.20%, -2.03%, and -1.45%, respectively. Compared with the Adaptive plan only, it lowers the training and test gaps by 0.85% and 1.24%, respectively; its test gap is also 0.75% lower than the Always-plan + component. These results support combining adaptive plan review with component evolution for both training performance and generalization.

Effect of Island Count and Plan Initialization. Table 7 compares three alternatives: 1) *Single island:* one plan with the full component library; 2) *Shared plan:* five islands initialized with the same plan and full library; 3) *Partitioned library:* five distinct plans generated from separate component-library partitions. Our default uses five distinct initial plans generated from the full library. It achieves the lowest training gap (-2.20%), improving on these alternatives by 0.89, 0.44, and 0.79 percentage points, respectively, supporting both initial-plan diversity and access to the full component library.

Table 7: Island initialization ablation.

Variant	Average gap (%)		
	Train	Val	Test
Single island	−1.31	−0.87	−0.45
Shared plan	−1.76	−1.29	−1.20
Partitioned library	−1.41	−1.87	**−1.51**
HeurEvo (Ours)	**−2.20**	**−2.03**	−1.45

We include the full table of ablation experiments in Appendix A.4.

5 Conclusion

We presented HeurEvo, a plan–code–component co-evolution framework for designing hybrid, solver-augmented heuristics for time-critical optimization. HeurEvo jointly evolves algorithmic structure, implementation, and reusable components through execution feedback, using plan-conditioned islands, adaptive plan review, and a shared component library. With a two-minute execution budget, the evolved heuristics outperform Gurobi's 48-hour solutions on seven of eleven synthetic tasks and several competitive LLM-based evolution baselines, with additional gains on MIPLIB and nonlinear problems. These results demonstrate the value of jointly evolving algorithmic structure and implementation rather than optimizing heuristic components in isolation.

AI USE STATEMENT

We used large language models (LLMs) in the proposed framework and experiments, to polish the authors' initial manuscript draft, and to generate code for constructing optimization instances. The authors reviewed the revised text, and the generated instances were checked against the experimental assumptions and requirements using human-written evaluators. Coding agents, including Codex and Claude Code, assisted with implementation, which human experts reviewed for consistency with the described framework. The authors take full responsibility for the manuscript and research artifacts, including the accuracy of claims and results, the validity of data and code, and the correctness and attribution of references.

REPRODUCIBILITY STATEMENT

To support reproducibility, we document the experimental setup, models, and implementation details in Section 4.1 and Appendix A, with experimental hyperparameters summarized in Table A8. Appendix B describes the data sources, generation and synthesis procedures, and dataset splits, while Appendix D provides the prompt templates. We will publicly release the code and datasets, including the data-generation scripts.

REFERENCES

Florian Arnold and Kenneth Sörensen. Knowledge-guided local search for the vehicle routing problem. *Computers & Operations Research*, 105:32–46, 2019.

Henrique Assumpção, Diego Ferreira, Leandro Campos, and Fabricio Murai. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

Albert-László Barabási and Réka Albert. Emergence of scaling in random networks. *Science*, 286 (5439):509–512, 1999. doi: 10.1126/science.286.5439.509. URL https://doi.org/10.1126/science.286.5439.509.

Timo Berthold, Dominik Kamp, Gioni Mexi, Sebastian Pokutta, and Imre Pólik. Global optimization for combinatorial geometry problems revisited in the era of LLMs, 2026a. URL https://arxiv.org/abs/2601.05943v2. Version 2. Table 1 gives squared distance ratios rounded upward to five decimal places.

Timo Berthold, Dominik Kamp, Gioni Mexi, Sebastian Pokutta, and Imre Pólik. Global optimization for combinatorial geometry problems revisited in the era of llms. *arXiv preprint arXiv:2601.05943*, 2026b.

Timo Berthold, Dominik Kamp, Gioni Mexi, Sebastian Pokutta, and Imre Pólik. Out-of-the-box global optimization for packing problems: New models and improved solutions, 2026c. URL https://arxiv.org/abs/2605.04850.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, et al. Adaevolve: Adaptive llm driven zeroth-order optimization. *arXiv preprint arXiv:2602.20133*, 2026.

Emilie Danna, Edward Rothberg, and Claude Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005. doi: 10.1007/s10107-004-0518-7.

G. B. Dantzig and J. H. Ramser. The truck dispatching problem. *Management Science*, 6(1):80–91, 1959. doi: 10.1287/mnsc.6.1.80.

Pham Vu Tuan Dat, Long Doan, and Huynh Thi Thanh Binh. Hsevo: Elevating automatic heuristic design with diversity-driven harmony search and genetic algorithm using llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 26931–26938, 2025.

Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

Cézar Augusto N e Silva and Haroldo Gambini Santos. Drawing graphs with mathematical programming and variable neighborhood search. *Electronic Notes in Discrete Mathematics*, 58:207–214, 2017.

Paul Erdős and Alfréd Rényi. On random graphs. I. *Publicationes Mathematicae Debrecen*, 6(3-4): 290–297, 1959. doi: 10.5486/PMD.1959.6.3-4.12. URL https://doi.org/10.5486/PMD.1959.6.3-4.12.

Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2(1):5–30, 1996. doi: 10.1007/BF00226291. URL https://doi.org/10.1007/BF00226291.

Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical programming*, 98(1):23–47, 2003.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale. *arXiv preprint arXiv:2511.02864*, 2025. doi: 10.48550/arXiv.2511.02864. URL https://arxiv.org/abs/2511.02864.

E. N. Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144, 1959. doi: 10.1214/aoms/1177706098. URL https://doi.org/10.1214/aoms/1177706098.

Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp M. Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, Marco Lübbecke, Hans D. Mittelmann, Derya Ozyurt, Ted K. Ralphs, Domenico Salvagnin, and Yuji Shinano. MIPLIB 2017: Data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, 13(3):443–490, 2021. doi: 10.1007/s12532-020-00194-3.

Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM*, 42(6):1115–1145, 1995. doi: 10.1145/227683.227684.

Bruce L. Golden, Larry Levy, and Rakesh Vohra. The orienteering problem. *Naval Research Logistics*, 34(3):307–318, 1987. doi: 10.1002/1520-6750(198706)34:3<307::AID-NAV3220340302>3.0.CO;2-D.

Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026. URL https://www.gurobi.com.

Jiachen Jiang, Tianyu Ding, and Zhihui Zhu. Deltaevolve: Accelerating scientific discovery through momentum-driven evolution. *arXiv preprint arXiv:2602.02919*, 2026.

Valentin Khrulkov, Andrey Galichin, Denis Bashkirov, Dmitry Vinichenko, Oleg Travkin, Roman Alferov, Andrey Kuznetsov, and Ivan Oseledets. Gigaevo: An open source optimization framework powered by llms and evolution algorithms. *arXiv preprint arXiv:2511.17592*, 2025.

Nguyen Viet Tuan Kiet, Tung Dao, Cong Dao Tran, and Huynh Thi Thanh Binh. Motif: Multi-strategy optimization via turn-based interactive framework. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 37000–37008, 2026.

Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E Bixby, Emilie Danna, Gerald Gamrath, Ambros M Gleixner, Stefan Heinz, et al. Miplib 2010: mixed integer programming library version 5. *Mathematical Programming Computation*, 3(2):103–163, 2011.

Alexey Kravatskiy, Valentin Khrulkov, and Ivan Oseledets. ImprovEvolve: Basin-hopping meets LLM-guided evolutionary search, 2026. URL https://arxiv.org/abs/2602.10233v2. Version 2, 26 June 2026. Version 1 was titled ImprovEvolve: Ask AlphaEvolve to Improve the Input Solution and Then Improvise.

Mingen Kuang, Xudong Deng, Xi Lin, Ye Fan, Jianyong Sun, and Jialong Shi. Llm-driven co-evolutionary automated heuristic design for bi-component coupled combinatorial optimization. *arXiv preprint arXiv:2606.00718*, 2026.

Gerald Lach and Marco E Lübbecke. Curriculum based course timetabling: new solutions to udine benchmark instances. *Annals of Operations Research*, 194(1):255–272, 2012.

Robert Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. In *International Conference on Learning Representations*, volume 2026, pp. 74026–74078, 2026.

Jonghwa Lee, Byung-In Kim, and Sang Hun Kim. A matheuristic algorithm for block assignment problem in long-term production planning in the shipbuilding industry. *Computers & Industrial Engineering*, 184:109603, 2023. doi: 10.1016/j.cie.2023.109603.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pp. 331–366. Springer, 2023.

Zhong Li, Hongliang Lu, Tao Wei, Yuxuan Chen, Wenyu Liu, Yuan Lan, Fan Zhang, and Zaiwen Wen. Constructing industrial-scale optimization modeling benchmark. *arXiv preprint arXiv:2602.10450*, 2026.

Fei Liu, Xialiang Tong, Mingxuan Yuan, and Qingfu Zhang. Algorithm evolution using large language model. *arXiv preprint arXiv:2311.15249*, 2023.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. Evolution of heuristics: Towards efficient automatic algorithm design using large language model. *arXiv preprint arXiv:2401.02051*, 2024.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z Pan, et al. Evox: Meta-evolution for automated discovery. *arXiv preprint arXiv:2602.23413*, 2026.

Yidong Luo, Xuemin Chen, Chenguang Wang, Fangzhou Zhu, Tao Zhong, and Tianshu Yu. Grimip: A general framework for instance-specific configuration of mip solvers using llms. *arXiv preprint arXiv:2606.23299*, 2026.

Haoze Lv, Ning Lu, Shengcai Liu, Shaofeng Zhang, and Ke Tang. Muevo: Llm-driven evolution of multi-heuristic ensemble. *arXiv preprint arXiv:2608.03636*, 2026.

Alan S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.

Silvano Martello and Paolo Toth. Lower bounds and reduction procedures for the bin packing problem. *Discrete Applied Mathematics*, 28(1):59–70, 1990.

C. E. Miller, A. W. Tucker, and R. A. Zemlin. Integer programming formulation of traveling salesman problems. *Journal of the ACM*, 7(4):326–329, 1960. doi: 10.1145/321043.321046.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

G. L. Nemhauser and L. E. Trotter, Jr. Vertex packings: Structural properties and algorithms. *Mathematical Programming*, 8:232–248, 1975. doi: 10.1007/BF01580444.

Jinbiao Nie, Kewei Feng, Xiaoyuan Zhang, Shan Yin, Zizhuo Wang, and Bin Dong. Milp-evo: Closed-loop fully automatic design of milp solvers. *arXiv preprint arXiv:2607.18252*, 2026.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. Introducing gpt-5.3-codex, 2026a. URL `https://developers.openai.com/api/docs/models/gpt-5.3-codex`.

OpenAI. GPT-5.6 Sol API. https://chatgpt.com, 2026b.

A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, 1969. doi: 10.1287/mnsc.16.1.93.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

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

Niki van Stein and Thomas Bäck. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *Trans. Evol. Comp*, 29(2):331–345, April 2025. ISSN 1089-778X.

E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64(2):278–285, 1993.

E-G Talbi. A taxonomy of hybrid metaheuristics. *Journal of heuristics*, 8(5):541–564, 2002.

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

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, et al. Thetaevolve: Test-time learning on open problems. *arXiv preprint arXiv:2511.23473*, 2025.

Yang Wu, Junran Pan, Yifan Zhang, Ning Xu, Fanshuo Zeng, and Jian Cheng. Refineevo: Planning-guided heuristic evolution with bidirectional experience. *arXiv preprint arXiv:2607.11358*, 2026.

Xianliang Yang, Ling Zhang, Haolong Qian, Lei Song, and Jiang Bian. Heuragenix: Leveraging llms for solving complex combinatorial optimization challenges. *arXiv preprint arXiv:2506.15196*, 2025.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. Multi-objective evolution of heuristic using large language model. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 27144–27152, 2025.

Danial Yazdani, Mohammad Nabi Omidvar, Yuan Sun, Maksud Ibrahimov, and Xiaodong Li. Atlas: Scaffold-free algorithm synthesis by llms via embedding-guided quality-diversity search. *arXiv preprint arXiv:2608.15546*, 2026.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. *Advances in neural information processing systems*, 37:43571–43608, 2024.

Zhi Zheng, Zhuoliang Xie, Zhenkun Wang, and Bryan Hooi. Monte carlo tree search for comprehensive exploration in llm-based automatic heuristic design. *arXiv preprint arXiv:2501.08603*, 2025.

A Additional Experiments

Implementation Details. We use GPT-5.3-Codex (OpenAI, 2026a) for the synthetic experiments and GPT-5.6-sol (OpenAI, 2026b) for both the MIPLIB-NL and nonlinear experiments. All experiments are conducted on a multi-node computing cluster, with each node equipped with 128 CPU cores and 1000 GB of memory. During evolution, each generated program is executed single-threaded on a single CPU core and is subject to a 120-second runtime limit per problem instance. Gurobi 13.0.2 is the only optimization solver permitted in the generated programs. All evolution experiments, including the baselines, are limited to generating 200 programs. For the synthetic experiments, generated programs are evaluated on the validation set during model selection. After evolution, the program with the best validation performance is selected and evaluated once on the held-out test set. Table A8 summarizes the complete hyperparameter configuration used for HeurEvo.

Table A8: Hyperparameters of HeurEvo. Symbols follow Appendix C; a dash marks a quantity the analysis does not name. Values are shared by all runs and ablation arms.

Symbol	Description	Value
Search budget and initialization		
K	Number of islands (one initial plan per island)	5
–	Validated seed programs generated per plan	1
M_{dbg}	Debugging attempts per generation call	3
B	Per-instance execution budget (seconds)	120
Island scheduling (Eqs. C1, C3)		
ρ	Discount factor of R, V, and H	0.9
C	UCB exploration coefficient	1.41
W	Local code visits per island at initialization (Alg. C1)	3
Code evolution (OpenEvolve backend)		
$\lvert\mathcal{I}_{j,h}^{(k)}\rvert$	Inspiration programs sampled per visit	2
–	Top-ranked programs shown in the prompt	2
–	Ancestor programs shown in the prompt (metrics only)	3
–	Elite / exploration / exploitation sampling ratio	0.1 / 0.2 / 0.7
–	MAP-Elites feature dimensions	complexity, diversity
–	MAP-Elites bins per dimension	10
–	Population size / archive size	1000 / 100
Plan evolution (Alg. C2)		
τ	Plateau threshold; $H_{j-1}^{(k)} < \tau$ triggers a plan review	0.01
W	Local code visits after an accepted plan proposal	3
–	Evidence: top programs under the current plan	2
–	Evidence: inspiration programs from the same island	3
–	Evidence: top programs from each other island	0
–	Evidence: inspiration programs across islands	5
Component evolution (Eq. C8)		
–	Rounds between Component Improver calls	1
C_I	Sensitivity to the contribution score $Q_j(s)$	2.0
$I_{\min}$	Minimum review probability	0.1
$I_{\max}$	Maximum review probability	0.7

A.1 Standard Synthetic problems

Tables A9 and A10 report all 11 problems on the training and test sets, respectively. EoH (Liu et al., 2024), ReEvo (Ye et al., 2024), and RefineEvo (Wu et al., 2026) retain separate constructive, ACO, and GLS variants wherever supported. In other words, their sparse task coverage prevents them from running all 11 tasks and excludes them from Table 2, which takes the mean objective gap

across all 11 problems. Among the full-coverage methods, HeurEvo achieves the lowest mean gap on both splits, with -2.22% on training and -1.45% on test, compared with -0.15% and $+0.91\%$ for AdaEvolve, respectively. This corresponds to reductions of 2.07 and 2.36 percentage points over AdaEvolve. At the individual-problem level, HeurEvo obtains the best or tied-best gap on six of the 11 problems on both training and test: C-CVRP, U-CVRP, OP, Decap, RCPS, and MIS. It also achieves negative gaps on eight training problems and seven test problems, showing that the gains are not confined to the training instances. The largest improvement over the full-coverage baselines occurs on OP, where HeurEvo changes the gap from $+16.92\%$ for AdaEvolve to -3.80% on training and from $+21.75\%$ to -1.94% on test.

Table A9: Full synthetic training gaps (%).

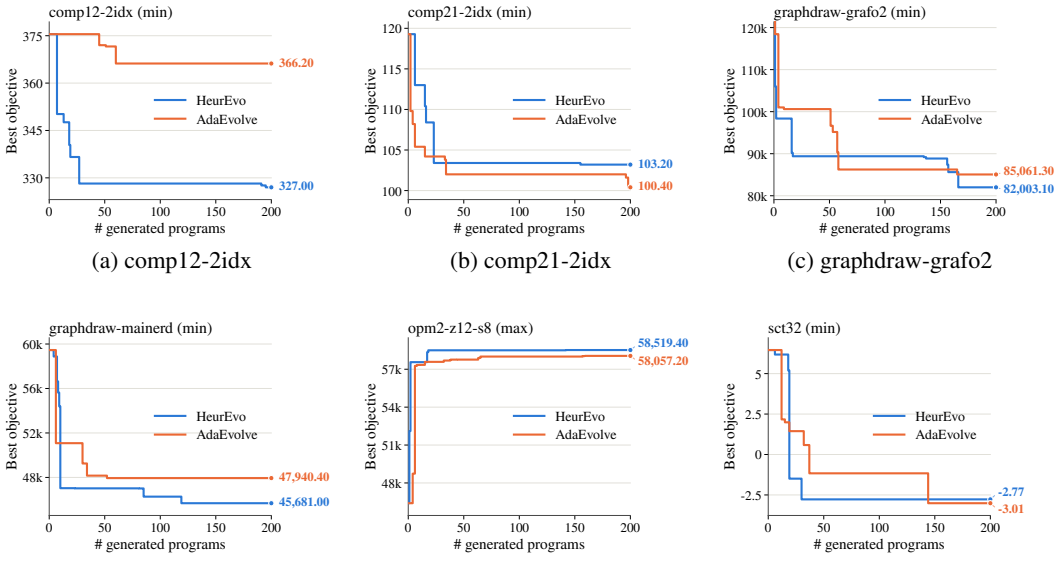
Method	C-CVRP	U-CVRP	TSP	BP	OP	Decap	JSS	PFSS	RCPS	Max-Cut	MIS
EoH (Constr.)	–	–	–	+3.53	–	–	–	–	–	–	–
EoH (GLS)	–	–	+0.68	–	–	–	–	**-2.65**	–	–	–
ReEvo (ACO)	+6.58	+9.66	+8.60	+1.14	+4.16	–	–	–	–	–	–
ReEvo (Constr.)	–	–	+16.02	–	–	–	–	–	–	–	–
ReEvo (GLS)	–	–	**-0.83**	–	–	–	–	–	–	–	–
RefineEvo (ACO)	+5.06	+9.10	+9.60	–	–	–	–	–	–	–	–
RefineEvo (Constr.)	+12.13	+16.10	+20.07	+2.29	–	–	–	–	–	–	–
RefineEvo (GLS)	–	–	**-0.83**	–	–	–	–	–	–	–	–
OpenEvolve	+1.93	-1.16	+1.53	+0.45	+16.65	+1.63	**-12.54**	-0.48	+0.64	-0.80	**-0.04**
AdaEvolve	-1.77	-2.06	-0.80	**-0.45**	+16.92	+1.63	-12.46	-2.19	+1.49	**-2.08**	+0.15
EvoX	+4.50	-1.63	+1.00	-0.20	+17.96	+1.50	-12.40	-0.30	+2.70	-0.17	+0.47
HeurEvo	**-3.36**	**-3.19**	-0.82	+0.45	**-3.80**	**+1.40**	-11.60	-2.25	**+0.12**	-1.32	**-0.04**

Table A10: Full synthetic test gaps (%).

Method	C-CVRP	U-CVRP	TSP	BP	OP	Decap	JSS	PFSS	RCPS	Max-Cut	MIS
EoH (Constr.)	–	–	–	+3.30	–	–	–	–	–	–	–
EoH (GLS)	–	–	+0.60	–	–	–	–	-2.20	–	–	–
ReEvo (ACO)	+8.28	+13.30	+9.96	+1.56	+7.40	–	–	–	–	–	–
ReEvo (Constr.)	–	–	+15.37	–	–	–	–	–	–	–	–
ReEvo (GLS)	–	–	**-0.79**	–	–	–	–	–	–	–	–
RefineEvo (ACO)	+8.18	+12.26	+12.72	–	–	–	–	–	–	–	–
RefineEvo (Constr.)	+9.91	+19.12	+23.22	+2.37	–	–	–	–	–	–	–
RefineEvo (GLS)	–	–	**-0.79**	–	–	–	–	–	–	–	–
OpenEvolve	+2.96	-1.30	+4.67	+0.69	+22.61	+2.04	-6.30	-0.27	+1.74	-0.31	**+0.14**
AdaEvolve	-1.62	-2.18	-0.38	**-0.32**	+21.75	+2.02	**-6.42**	**-2.21**	+0.70	**-1.78**	+0.45
EvoX	+2.17	-0.20	+3.14	-0.06	+21.57	+1.95	-6.16	-0.71	+0.91	+0.11	+1.88
HeurEvo	**-4.38**	**-3.48**	-0.70	+0.69	**-1.94**	**+1.90**	-5.52	-1.85	**0.00**	-0.85	**+0.14**

A.2 MIPLIB-DERIVED PROBLEMS

Earlier in Table 4, we report the averaged training objectives on MIPLIB problems. Figure 4 demonstrates the training performance of each method, i.e., mean objectives vs. the number of generated programs. The trajectories reveal different improvement patterns across problems. On `comp12-2idx`, `graphdraw-mainerd`, and `opm2-z12-s8`, HeurEvo establishes an advantage within the first few dozen generated programs and maintains it through the end of the run. On `graphdraw-grafo2`, AdaEvolve leads during the middle of training, but late-stage improvements allow HeurEvo to finish with a lower objective of 82003.10, compared with 85061.30 for AdaEvolve. However, HeurEvo does not dominate on every problem. AdaEvolve achieves a lower final objective on `comp21-2idx`, reaching 100.40 compared with 103.20 for HeurEvo. On `sct32`, HeurEvo reaches an objective near -2.77 within roughly 30 generated programs but makes little subsequent progress, whereas AdaEvolve overtakes it after approximately 140 programs and



(a) comp12-2idx (b) comp21-2idx (c) graphdraw-grafo2

(d) graphdraw-mainerd (e) opm2-z12-s8 (f) sct32

Figure 4: Best objective versus number of generated programs for HeurEvo and AdaEvolve (solver model GPT-5.6) on the six MIPLIB instances. Objectives are averaged over the five training seeds; the value at the end of each curve is the final best objective.

finishes at -3.01. These contrasting trajectories show that early advantages do not always determine the final ranking and that improvements after extended plateaus can remain important.

A.3 Non-linear Geometry Problems

Table A11 contains all 14 nonlinear task-size configurations under GPT-5.6-sol, which is extended from the main Table 5. In this table, we offer rich information; for example, we show the output of each best-known solution. Moreover, we demonstrate the performance where the proposed HeurEvo achieves poorer performance. Across all 14 configurations, HeurEvo achieves better objectives than AdaEvolve in 11, matches it at the reported precision in one, and performs worse in two. Its improvements cover all three rectangle-packing configurations, all four distance-ratio configurations, and four of the five hexagon-packing configurations. However, these improvements vary in magnitude. For example, on hexagon packing with $n = 11$, HeurEvo reduces the AdaEvolve objective to 3.9245011309 at the second decimal place, approaching the best-known solution. In contrast, its improvement over AdaEvolve for $n = 16$ is only 1.812×10^{-7}, and the resulting objective of 4.5324919358 remains slightly above the reported reference.

A.4 Full Ablation Results on Synthetic Problems

Earlier in Table 6, we show the effectiveness of various components by showing the mean objective gap across all 11 synthetic tasks and choose the four most representative problems to analyze. Tables A12 and A13 extend from Table 6 with all 11 syntheticc tasks performance. Across all 11 tasks, HeurEvo achieves the lowest gap on five tasks in each split, leading on C-CVRP, U-CVRP, TSP, and PFSS in both training and testing. It outperforms *Adaptive plan only* on nine tasks and *Component only* on eight tasks in each split. Compared with *Always-plan + component*, it achieves lower gaps on seven training tasks and six test tasks. These results support combining adaptive plan review with component evolution for both training performance and generalization.

Table A11: Complete nonlinear training objectives. For HeurEvo, Underline indicates the part better than AdaEvolve, and **Bold** indicates the part better than the reported best-known value.

Problem	n	d	Best result source	Reported reference	AdaEvolve	HeurEvo
Circle packing (rectangle) ↑	21	–	EinsteinArena[a]	2.3658323759185156	2.3642481267	2.3658323759
Circle packing (rectangle) ↑	26	–	Lai[b]	2.639320558987759336	2.6355313151	2.639320**5643**
Circle packing (rectangle) ↑	27	–	Dutton / Lai[b,c]	2.691523360671018652	2.6834410277	2.6891174195
Circle packing (square) ↑	26	–	Packomania[d]	2.635983084919	2.6359830484	2.6359775049
Circle packing (square) ↑	32	–	Berthold et al. (2026a)[d]	2.939572771205	2.9395727706	2.9395727706
Distance ratio ↓	14	3	Sun–Samanta[e]	4.165	4.1657834746	4.165782**0926**
Distance ratio ↓	16	2	Together-AI[f]	12.8892299	12.8892299077	12.889229**1072**
Distance ratio ↓	21	2	Berthold et al. (2026a)	17.77499	17.7749805542	17.774980**3417**
Distance ratio ↓	22	2	Berthold et al. (2026a)	19.05398	19.0539845098	19.053976**9136**
Hexagon packing ↓	11	–	Schellhorn[g]	3.9245008972987525	3.9326758559	3.924501**1309**
Hexagon packing ↓	12	–	Berthold et al. (2026c)[h]	3.941642067536949	3.9416423004	3.941642**0625**
Hexagon packing ↓	14	–	Berthold et al. (2026c)[h]	4.268994957252001	4.2723920772	4.268995**0806**
Hexagon packing ↓	15	–	Kravatskiy et al. (2026)[g,h]	4.447271148987105	4.4472718765	4.4494139689
Hexagon packing ↓	16	–	Kravatskiy et al. (2026)[g]	4.52746	4.5324921170	4.532491**9358**

[a] EinsteinArena solution and verification notebook: `https://github.com/togethercomputer/EinsteinArena-new-SOTA/blob/c388c6f7408c886311940896713339a1a70c2394/circles-rectangle/analysis.ipynb`.
[b] Lai's public rectangle configurations (files `coords_n26.txt` and `coords_n27.txt`): `https://github.com/lshhhhhhh/circle-packing-records/tree/8dfa6449385015a58aabfbf84f9dd5f063bfa96b`.
[c] Friedman's rectangle record list (including Dutton's $n = 27$ record): `https://erich-friedman.github.io/packing/cirRrec/`.
[d] Packomania's circle-in-square numerical records: `https://www.packomania.com/csqv/csqv.html`.
[e] Friedman's three-dimensional distance-ratio records: `https://erich-friedman.github.io/packing/maxmin3/`.
[f] Together-AI's reported score on the EinsteinArena leaderboard for the two-dimensional distance-ratio problem ($n = 16$): `https://einsteinarena.com/problems/min-distance-ratio-2d`.
[g] Friedman's hexagon-in-hexagon records: `https://erich-friedman.github.io/packing/hexinhex/`.
[h] Public numerical configurations accompanying the packing paper (subdirectories `n12`, `n14`, and `n15`): `https://github.com/DominikKamp/Packing/tree/27f71632d8655ae4b6a049410a401976748b94e3/polygon/l6/m6`.

Table A12: Synthetic ablation training gaps (%).

Method	C-CVRP	U-CVRP	TSP	BP	OP	Decap	JSS	PFSS	RCPS	Max-Cut	MIS
Fixed plan	-3.15	-2.25	+1.23	+0.45	+1.94	+1.47	-12.29	-0.83	+1.39	-1.19	+0.19
Adaptive plan only	-2.48	-0.68	+1.11	+0.45	-0.87	+1.43	**-12.32**	-1.48	+0.87	-1.18	+0.27
Component only	-3.09	-2.47	+1.98	+0.40	-1.67	**+1.30**	-12.28	-0.79	+1.37	-1.08	+0.35
Earlier full configuration	-3.04	-1.82	+1.45	+0.45	-0.75	+1.45	**-12.32**	-1.15	+0.82	-1.83	+0.23
Always-plan + component	-3.20	-1.78	+0.40	+0.45	-2.67	+1.47	**-12.32**	-1.16	**-0.35**	-1.69	+0.11
Always-plan	-2.84	-2.04	-0.23	**+0.10**	+0.45	+1.44	-12.23	-1.52	+0.50	**-1.89**	**-0.12**
HeurEvo (Ours)	**-3.36**	**-3.19**	**-0.82**	+0.45	**-3.80**	+1.40	-11.60	**-2.25**	+0.12	-1.32	-0.04

Table A13: Synthetic ablation test gaps (%).

Method	C-CVRP	U-CVRP	TSP	BP	OP	Decap	JSS	PFSS	RCPS	Max-Cut	MIS
Fixed plan	-3.26	-1.81	+2.53	+0.69	+7.97	+2.66	-6.30	-0.50	+0.39	-0.86	+1.20
Adaptive plan only	-3.00	-1.81	+2.44	+0.69	+3.76	+2.01	**-6.57**	-0.64	+0.65	-0.79	+0.90
Component only	-3.47	-1.59	+4.17	+0.69	+3.86	+1.62	-6.41	-0.69	+1.61	-0.70	+1.79
Earlier full configuration	-3.45	-1.06	+1.78	+0.69	+7.04	+1.98	-6.42	-0.89	+0.57	**-1.54**	+0.57
Always-plan + component	-3.62	+0.09	+1.97	+0.69	**-2.39**	**+1.82**	-6.19	-0.93	+0.43	-1.08	+1.52
Always-plan	-2.95	-1.96	+0.18	**+0.38**	+8.12	+2.03	-6.17	-1.67	+0.43	-1.44	**+0.11**
HeurEvo (Ours)	**-4.38**	**-3.48**	**-0.70**	+0.69	-1.94	+1.90	-5.52	**-1.85**	**0.00**	-0.85	+0.14

A.5 ISLAND INITIALIZATION ABLATIONS

Tables A14 and A15 report objective gaps of every problem for the *Single island*, *Shared plan*, and *Partitioned library* variants in Table 7. Compared with *Single island*, *Shared plan* achieves lower test gaps on six tasks, including a reduction from $+2.09\%$ to -2.96% on U-CVRP, showing that multiple islands can help even when initialized with the same plan. With the island count fixed at five, HeurEvo uses distinct initial plans and improves upon *Shared plan* on seven validation tasks. It also lowers the test gaps on U-CVRP, OP, and RCPS, with the OP gap decreasing from $+3.85\%$ to -1.94%. Although individual tasks favor different configurations, these results support the importance of both the number of islands and the diversity of their initial plans.

Table A14: Island ablation validation gaps (%). Bold indicates the lowest gap in each column, including ties.

Variant	C-CVRP	U-CVRP	TSP	BP	OP	Decap	JSS	PFSS	RCPS	Max-Cut	MIS
Single island	-2.88	+3.10	-0.92	+0.49	-0.06	**+1.05**	-7.07	**-2.65**	-0.57	-1.32	+0.10
Shared plan	-2.69	-1.93	**-0.98**	+0.62	-0.97	+1.55	-6.61	-2.35	+1.21	**-1.41**	**-0.10**
Partitioned library	-1.88	-1.94	-0.93	+0.49	**-3.98**	+1.17	-6.74	-1.86	**-1.78**	-1.23	+0.10
HeurEvo (Ours)	**-3.36**	**-3.19**	-0.82	**+0.45**	-3.80	+1.40	**-11.60**	-2.25	+0.12	-1.32	-0.04

Table A15: Island ablation test gaps (%). Bold indicates the lowest gap in each column, including ties.

Variant	C-CVRP	U-CVRP	TSP	BP	OP	Decap	JSS	PFSS	RCPS	Max-Cut	MIS
Single island	**-4.87**	+2.09	-0.79	**+0.50**	+4.18	+1.20	-6.90	**-2.31**	+0.29	**-1.60**	0.00
Shared plan	-4.55	-2.96	**-0.84**	**+0.50**	+3.85	**+1.06**	**-7.46**	-2.05	+0.58	-1.42	**-0.26**
Partitioned library	-3.58	**-3.74**	-0.78	**+0.50**	-1.46	+1.45	-6.90	-1.52	+0.29	-1.51	-0.07
HeurEvo (Ours)	-4.38	-3.48	-0.70	+0.69	**-1.94**	+1.90	-5.52	-1.85	**0.00**	-0.85	+0.14

B Dataset Details

B.1 Dataset Composition

We consider three data types: Synthetic Problems, MIPLIB-Derived Problems, and Nonlinear Geometry Benchmarks. The first two form the instance collections summarized below; the nonlinear configurations are described in Section B.4. In these two collections, a task denotes one configured synthetic problem or one MIPLIB-NL base problem together with its perturbations. The 11 synthetic tasks represent 10 mathematical problem families because capacitated vehicle routing has two configurations. The six MIPLIB-derived tasks cover four application domains.

Table B16 counts the instances used in the reported experiments. Each synthetic task uses 11 instances: `seed0–seed4` for training, `seed5–seed6` for validation, and `seed7–seed10` for testing. The MIPLIB experiments reported here use five training instances per task, `seed0–seed4`. These are instance seeds, rather than independent seeds of heuristic-evolution runs. Section B.2 qualifies the effective diversity of the decap-placement task.

Instance Format. All our data follow the MIPLIB-NL format (Li et al., 2026). Each instance includes a natural-language problem description, large input data supplied as CSV tables, and an initial Gurobi program encoding the mathematical formulation. A standardized solution-output format and a task-specific validator allow generated programs to be evaluated consistently: the validator checks the output solution's feasibility and the consistency of its reported objective.

Table B16: Instances used in the reported experiments. Dashes indicate splits outside the reported evaluation.

Collection	Tasks	Train	Val.	Test	Total
Synthetic Problems	11	55	22	44	121
MIPLIB-Derived Problems	6	30	–	–	66
Total	17	85	22	44	151

Data Availability. We will soon open-source both our synthetic dataset and our MIPLIB-NL-derived dataset, together with the generation code, instance specifications, reference models, and split definitions.

Fitness Score Function Definition. We use task-specific fitness functions to evaluate candidate programs. As defined earlier in Section 3.1, $\mathcal{D}$ denotes the training set for a target task, and $d \in \mathcal{D}$ represents an individual instance. Let $\mathrm{obj}(G; d)$ denote the verified objective value of the feasible solution returned by program G on instance d within the runtime budget B. For the synthetic and MIPLIB-derived problems (Tables B17 and B18), we define

$$f(G; d) = -\frac{|\mathrm{obj}(G; d) - b_{48\mathrm{h}}(d)|}{|\mathrm{obj}(G; d)| + \epsilon}, \tag{2}$$

where $b_{48\mathrm{h}}(d)$ is the best objective bound obtained from a Gurobi reference run with a 48-hour time limit. This is a lower bound for minimization problems and an upper bound for maximization problems, rather than the objective value of Gurobi's best feasible solution. The bound is fixed for each instance across program evaluations, and $\epsilon > 0$ prevents division by zero.

For the nonlinear geometry problems (Section B.4), we directly transform the objective according to its optimization direction,

$$f(G; d) = -s(d)\,\mathrm{obj}(G; d), \qquad s(d) = \begin{cases} +1, & \text{for minimization,} \\ -1, & \text{for maximization.} \end{cases} \tag{3}$$

Evolution maximizes the mean training fitness, $F(G) = |\mathcal{D}|^{-1} \sum_{d \in \mathcal{D}} f(G; d)$. Each nonlinear configuration is optimized separately, so its training set contains a single instance. These fitness scores guide program evolution and are distinct from the signed gaps relative to Gurobi's 48-hour incumbent reported in the experimental results.

B.2 SYNTHETIC PROBLEMS

Table B17 describes the 11 synthetic tasks and how we generate their instances. We follow published sampling recipes where noted and specify our changes in the corresponding entries. Problem sizes are fixed within a task; realized edge counts, precedence counts, and demand-derived fleet sizes can vary. Numerical ranges below are measured across all 11 stored instances per task. Integer-uniform ranges include both endpoints. For TSP, both CVRP configurations, and orienteering, coordinates are stored to six decimal places, and distances between stored points p_i and p_j are

$$d_{ij} = \mathrm{round}(1000\|p_i - p_j\|_2)\,. \tag{4}$$

Decap placement requires a further qualification. Its randomized target weights cancel against inversely scaled base requirements, leaving essentially the same optimization model across seeds, apart from small rounding differences. Its nominal split therefore does not measure generalization to substantively different instances.

Table B17: Synthetic problem definitions and generation settings.

Task	Definition, generation details, and references
Clustered CVRP	Minimize total distance while serving each customer exactly once using exactly 22 depot-return routes, each carrying at most 55 demand units. There are 120 customers in four Gaussian clusters with coordinate standard deviation 0.08, clipped to the unit square. The depot is at $(0.1, 0.1)$; customer demands are uniform integers from 1 to 13. Each customer independently chooses a cluster center uniformly from $(0.25, 0.25)$, $(0.75, 0.30)$, $(0.35, 0.78)$, and $(0.78, 0.75)$ before its coordinates are sampled. This is our custom clustered generator for the CVRP of Dantzig & Ramser (1959), not a reproduction of a published instance set.
Uniform CVRP	Minimize total distance with complete single-visit service, depot-return routes, and vehicle capacity 50. The 200 customers are sampled uniformly from the unit square, with a depot at $(0.5, 0.5)$ and demands uniform on the integers 1–9. The fleet size is fixed for each instance and ranges from 20 to 22 vehicles: $$K = \max\left(\left\lceil \frac{\sum_i q_i}{50} \right\rceil + 1, K_{\mathrm{FFD}}\right),$$ where K_{FFD} is the number of capacity-50 bins obtained by first-fit decreasing on the demands. Coordinates, demands, depot, and capacity follow the ACO benchmark sampling of ReEvo (Ye et al., 2024); the exact fleet-size constraint and integer distances are our adaptations.
Decap-Placement Proxy	Minimize capacitor cost subject to placing at most one capacitor type at each site and meeting every target's required cumulative synthetic effect. A 30×30 grid supplies 900 sites and 900 targets. Four types give 3,600 site–type choices; effects decay with distance inside a type-specific radius. Type k contributes $e_k/(1 + d_{st})$ when the Euclidean grid distance $d_{st} \le \rho_k$, and zero otherwise. The four $(\mathrm{cost}, e_k, \rho_k)$ tuples are $(1.0, 1.4, 1.25)$, $(2.1, 2.6, 2.25)$, $(3.8, 4.3, 3.5)$, and $(6.4, 7.0, 5.25)$. Before rounding, target t requires $0.22 m_t F_t$, where F_t is the strongest-type all-site coverage and $m_t = 0.90 + 0.20\|p_t - (14.5, 14.5)\|_2/(14.5\sqrt{2})$. Target-weight jitter is uniform on $[-0.06, 0.06]$ but cancels against inversely scaled base requirements, up to rounding. This covering proxy is inspired by ReEvo's application (Ye et al., 2024), not its electrical simulator or data distribution; see Section B.2 for the diversity qualification.

Continued on next page

Table B17 (continued)

Task	Definition, generation details, and references
Job-Shop Scheduling	`job_shop_scheduling`. Minimize makespan while preserving each job's operation order and preventing machine conflicts. Operations cannot be interrupted. There are 100 jobs and 20 machines (2,000 operations). Every job visits every machine once in a random order, with processing times uniform on the integers 1–99. This follows Taillard's random-permutation and processing-time recipe (Taillard, 1993), with fresh seeds; the scheduling formulation follows Manne (1960).
Max-Cut	`max_cut`. Partition the vertices into two sets to maximize the number of crossing edges (Goemans & Williamson, 1995). Each graph has 2,000 vertices and 10,882–11,072 edges. Our hybrid generator combines Barabási–Albert-style preferential attachment (Barabási & Albert, 1999), initialized with a five-vertex clique and attachment parameter 4, with an independent Erdős–Rényi $G(n, p)$ edge overlay at $p = 0.0015$ (Erdős & Rényi, 1959; Gilbert, 1959). Duplicate edges are merged; all stored edge weights equal one. This is our custom combination of the two graph models.
Maximum Independent Set	`maximum_independent_set`. Maximize the cardinality of a vertex subset containing no adjacent pair. Each preferential-attachment graph has 1,500 vertices and 11,964 edges, uses attachment parameter 8, and starts from a nine-vertex clique. We follow the preferential-attachment principle of Barabási & Albert (1999) with this explicit initialization and use the unweighted independent-set objective (Nemhauser & Trotter, 1975).
Offline Bin Packing (BP)	`offline_bin_packing`. Minimize the number of bins while assigning each item to exactly one bin without exceeding capacity. All items are available before packing (Martello & Toth, 1990). Each instance has 1,000 items with independently sampled uniform integer sizes from 20 to 100 and bin capacity 150, following Falkenauer's uniform benchmark recipe (Falkenauer, 1996). These are newly generated instances, not the original benchmark files.
Orienteering	`orienteering`. Maximize the prize collected on one depot-return tour visiting a subset of customers within a travel-distance budget. The 500 nodes comprise a centered depot and 499 uniformly distributed customers. The distance budget is $\mathrm{round}(1000 \cdot 0.4\sqrt{500}) = 8944$. Customer prizes are $$\pi_i = \mathrm{clip}_{[1,100]}(\mathrm{round}(15 + 75r_i + \epsilon_i)),$$ where $\epsilon_i \sim \mathcal{N}(0, 8^2)$ and r_i is the distance from the centered depot divided by $\sqrt{0.5}$, computed before coordinate rounding. The distance-correlated prize principle follows ReEvo (Ye et al., 2024); noise, normalization, centered depot, and budget are our adaptations to the orienteering problem (Golden et al., 1987).

Continued on next page

Table B17 (continued)

Task	Definition, generation details, and references
Permutation Flow-Shop Scheduling	`permutation_flow_shop_scheduling`. Minimize makespan. All jobs follow the same machine order, and all machines process jobs in the same chosen permutation. Operations of a job cannot overlap, and each machine processes at most one job at a time.
	The 100 jobs and 20 machines have 2,000 independently sampled uniform integer processing times from 1 to 100. This is a Taillard-style synthetic recipe (Taillard, 1993), using an upper endpoint of 100 rather than Taillard's 99; we do not reuse the published benchmark instances.
Resource-Constrained Project Scheduling	`resource_constrained_project_scheduling`. Minimize makespan subject to finish–start precedence and renewable-resource capacities during each activity's uninterrupted execution.
	Instances have 85 activities, five resources, and 212–265 precedence arcs, with no dummy activities. Durations are uniform integers from 1 to 8 and capacities from 5 to 12. The first resource has comparatively larger capacity-relative demands: activity demands are uniform integers from zero to $\lfloor C_0/2 \rfloor$ for resource zero and to $\lfloor C_r/3 \rfloor$ otherwise.
	Each pair $i < j$ receives an arc with probability 0.06; consecutive pairs receive an independent additional opportunity with probability 0.35, with duplicates merged. This is our custom acyclic-network recipe for the classical formulation (Pritsker et al., 1969), not a replication of PSPLIB generation.
Traveling Salesman	`traveling_salesman`. Minimize the length of a Hamiltonian cycle visiting every node exactly once and returning to its start.
	Each instance contains 350 nodes drawn from three Gaussian clusters with coordinate standard deviation 0.08 and clipped to the unit square. Each node independently chooses one of $(0.2, 0.25)$, $(0.78, 0.24)$, and $(0.5, 0.75)$ with equal probability. This is our custom clustered Euclidean generator; the reference model uses the subtour formulation of Miller et al. (1960).

B.3 MIPLIB-Derived Problems

Table B18 describes the six MIPLIB-derived tasks. MIPLIB-NL (Li et al., 2026) reconstructs natural-language optimization problems from MIPLIB 2017 (Gleixner et al., 2021). We selected these six base problems because Gurobi (Gurobi Optimization, LLC, 2026) did not certify optimality within two hours (7,200 seconds) per instance. This selection criterion refers to the initial screening, not subsequent reference re-solves or a claim about all instances in MIPLIB-NL. The tasks comprise two curriculum-based course-scheduling problems (`comp12-2idx`, `comp21-2idx`) (Lach & Lübbecke, 2012), two graph-drawing problems (`graphdraw-grafo2`, `graphdraw-mainerd`) (e Silva & Santos, 2017), mining-project selection (`opm2-z12-s8`), and PCB assembly-line configuration (`sct32`) from MIPLIB (Koch et al., 2011). We use these original IDs consistently in the dataset and result tables. For each task, the five training instances comprise the unperturbed reconstructed base instance (`seed0`) and four task-specific perturbations (`seed1–seed4`). The dimensions below are fixed across these training instances.

Table B18: MIPLIB-derived problem definitions.

Problem ID	Definition and instance configuration
`comp12-2idx`	`Coursescheduling`. Schedule all required lectures into available periods while avoiding course conflicts and respecting the number of rooms. Minimize penalties for room-capacity shortfalls, insufficient teaching-day spread, and isolated curriculum lectures. There are 88 courses, 218 lectures, six days with six periods each, 11 rooms, 146 curricula, and 519 course-conflict pairs.
`comp21-2idx`	`Coursescheduling`. Solve the same time-assignment problem and three-part penalty objective, with a different timetable, room inventory, and curriculum structure. Room capacities are represented by capacity thresholds and penalized shortfalls. There are 94 courses, 327 lectures, five days with five periods each, 18 rooms, 78 curricula, and 302 conflict pairs.
`graphdraw-grafo2`	`Graphdrawing`. Place entities at integer coordinates inside their permitted windows, enforcing size-dependent non-overlap and consistent directional relations. Minimize weighted connection-distance terms plus deviation from preferred anchors. This is the entity-placement phase of graph drawing, with 67 entities, 73 connections, and 2,211 entity pairs. All edge weights equal 122.
`graphdraw-mainerd`	`Graphdrawing`. Solve the same entity-placement problem on a smaller graph, with placement windows, pairwise separations, directional consistency, and centering terms. There are 31 entities, 33 connections, and 465 entity pairs. All edge weights equal 126.
`opm2-z12-s8`	`Mining-projectselection`. Select mining blocks/projects to maximize total net present value. Selecting a project requires selecting its predecessors, and the total use of each resource must respect its capacity. There are 10,800 binary project decisions, eight resources, and 319,500 dependency arcs. Every resource has capacity 3,704,832.
`sct32`	`PCBassembly-lineconfiguration`. Assign each operation's full workload across eligible workstation options, using binary decisions on discrete lanes and fractional assignments where permitted. Minimize staffing cost minus weighted checkpoint performance and discrete-routing rewards, subject to capacities, checkpoint constraints, priority-token eligibility and budget, and policy checks. There are 552 operations, ten workstation options, seven board variants, 27 checkpoints, 67 policy checks, and a priority-token budget of six.

B.3.1 Generating MIPLIB Training Instances

We construct the five training instances for each of the six reconstructed base problems as follows. For each problem, `seed0` retains the base data, and `seed1`–`seed4` vary selected relationships and coefficients while preserving table dimensions, course–capacity keys, or graph degree sequences, as appropriate. For `sct32`, we use the upstream generator's numerical-jitter mode. Across all six tasks, the variable count remains unchanged; linear constraint counts may vary for timetabling (Table B19).

Course Timetabling. For both tasks, we relocate unavailable periods while preserving their total count for each course. We first sample full-day unavailability and then remaining slots without replacement, using the base instance's corresponding day and day–period frequencies plus one as sampling weights. We retain the existing curriculum-induced conflict pairs and apply degree-preserving double-edge swaps to the remaining conflicts: 83 pairs for `comp12-2idx` and 32 for `comp21-2idx`. We also permute lecture-count/minimum-day pairs across courses, preserving their multiset and total lecture count. A permutation is accepted only if each course's lecture count fits its available periods and each curriculum's lecture load does not exceed the base instance's maximum curriculum load. Finally, we permute capacity-penalty weights within each capacity class. Curricula, room inventories, course identifiers, and course–capacity penalty keys are retained.

Graph Drawing. Degree-preserving edge swaps modify the connection graph while retaining entity data, placement windows, sizes, edge count, degree sequence, and the constant edge weight. Edge-distance coefficients are then recomputed from the unchanged geometry.

Mining-Project Selection. Dependency arcs and resource capacities are retained. Project values and resource costs are independently multiplied by factors from $U(0.9, 1.1)$ and rounded to integers; stored resource-cost entries are bounded below by one, while omitted entries remain zero. Thus, dependency structure is preserved, while changes to resource coefficients can change the feasible set.

PCB Assembly-Line Configuration. The upstream generator's jitter mode independently multiplies entries in designated numeric data-table columns by factors from $U(0.9, 1.1)$, using floating-point conversion and nonnegative clipping without integer rounding. These columns contain capacities, workloads, performance coefficients, weights, and constraint right-hand sides. Identifiers, categorical flags, table structure, and scalar model parameters are retained.

Feasibility Checks. We check the timetable variable counts against an explicit counting formula and verify that the graph-drawing coefficient formulas reproduce the base data. We then construct each instance's Gurobi model to record its size before optimization, obtain a reference solution, and pass the reported solution to the task-specific verifier to check feasibility and objective consistency. Generation-time reference runs used a 120-second budget with two threads, with additional 600-second, four-thread runs for `graphdraw-grafo2`. The generation-time verification records contain a `VALID` verdict for each of the 30 training instances; these checks are separate from both the initial hardness screening and subsequent longer reference re-solves. The validated reference objectives are feasible-solution values; a successful verification does not certify optimality. The timetabling guards are screening conditions, not a guarantee of feasibility. If a draw is proven infeasible, we resample it using a new random-stream attempt index and repeat validation.

Seeds, resampling attempts, and perturbation details are recorded with each instance, together with its feasibility-check results.

B.3.2 MIPLIB Model Sizes

Table B19 reports the numbers of decision variables and constraints in the Gurobi models used for the six MIPLIB-derived problems. Each problem has the same number of variables across its 11 instances. General constraints are reported separately from linear constraints.

Table B19: Formulation sizes before presolve for 11 synthetic and six MIPLIB-derived tasks. Ranges cover synthetic instances; "varies" marks MIPLIB constraint counts affected by perturbation.

Problem	Variables	Linear constraints	General constraints	Key instance parameters
C-CVRP	14,640	14,522	0	120 customers; 22 vehicles; capacity 55

Continued on next page

Table B19 (continued)

Problem	Variables	Linear constraints	General constraints	Key instance parameters
U-CVRP	40,400	40,202	0	200 customers; 20–22 vehicles; capacity 50
OP	250,498	250,501	0	500 nodes; travel budget 8,944
TSP	122,499	122,152	0	350 nodes
JSSP	101,001	201,900	0	100 jobs; 20 machines; 2,000 operations
PFSSP	12,000	4,081	0	100 jobs; 20 machines
RCPSP	28,646–34,358	2,140–2,449	0	85 activities; 5 resources; 212–265 precedences
BP	1,001,000	2,000	0	1,000 items; bin capacity 150
`comp12-2idx`	11,726	varies	0	88 courses; 36 time slots; 11 rooms; 519 conflicts
`comp21-2idx`	10,911	varies	0	94 courses; 25 time slots; 18 rooms; 302 conflicts
Max-Cut	12,882–13,072	43,528–44,288	0	2,000 vertices; 10,882–11,072 edges
MIS	1,500	11,964	0	1,500 vertices; 11,964 edges
`graphdraw-grafo2`	9,258	194,611	9,136	67 entities; 73 relationships; 2,211 entity pairs
`graphdraw-mainerd`	2,050	18,801	1,992	31 entities; 33 relationships; 465 entity pairs
DCP	3,600	1,800	0	900 sites; 900 targets; 4 capacitor types
`opm2-z12-s8`	10,800	319,508	0	10,800 projects; 8 resources; 319,500 dependencies
`sct32`	6,607	3,885	0	552 operations; 10 stations; 7 variants; 27 checkpoints

B.4 Nonlinear Geometry Benchmarks

We take our nonlinear geometry benchmarks from Berthold et al. (2026b), who revisit problems studied in AlphaEvolve (Georgiev et al., 2025; Novikov et al., 2025) by solving those with nonlinear programming solvers. We retain these AlphaEvolve problem definitions and use the nonlinear formulations presented by Berthold et al. (2026b). We manually wrote our own natural-language problem descriptions and mathematical formulation files for these settings. We evolve heuristics separately for each problem–size configuration.

Circle packing maximizes the sum of variable radii in either a unit square or a rectangle of perimeter four with an optimized aspect ratio. The distance-ratio problem minimizes the ratio of maximum to minimum pairwise distances between points; our formulation uses squared distances and fixes the minimum squared distance to one. Hexagon packing minimizes the side length of a regular hexagonal container holding non-overlapping unit-side regular hexagons with free translations and rotations.

Table B20 lists all 14 configurations in the nonlinear collection. Here, n denotes the number of circles, points, or hexagons, and d denotes the dimension of the point configuration.

Table B20: Nonlinear problem names and size configurations.

Problem Name	Size Configurations
`circle_packing_square`	$n \in \{26, 32\}$
`circle_packing_rectangle`	$n \in \{21, 26, 27\}$
`distance_ratio`	$d = 2,\ n \in \{16, 21, 22, 29\}$; $d = 3,\ n = 14$
`hexagon_packing`	$n \in \{11, 12, 14, 15, 16\}$

These fixed geometry configurations do not use separate instances for training, validation, and testing. Seed directories identify problem sizes rather than random samples. These benchmarks are additional to the 151 instances used in the reported experiments in Table B16.

C Additional Method Details

C.1 Notations

The indices k, j, h, i identify the island, global round, local update, and program step, respectively. The tag role is seed, par, child, or $\star$. Omit h when no local update is involved. For shared best programs, scope is global or archive.

Table C1: Annotations

Annotation	Description
$\mathcal{A}, \mathcal{A}_j$	Shared Program Database of retained, validated programs.
$\mathcal{A}_j^{(k)}$	All retained programs from island k, including earlier plans.
$\mathcal{A}_j^{(k,\mathrm{par})}, \mathcal{C}_j^{(k)}$	Parent-eligible programs under the current plan generation.
$\mathcal{A}_j^{(k,\mathrm{ins})}$	Available inspiration programs for island k.
$\mathcal{D}, B$	Training set and per-instance execution budget.
$f(G;d), F(G)$	Instance fitness and average training fitness; larger is better.
$F_{j,h}^{(k,\mathrm{role})}$	Fitness of the correspondingly indexed program G; shared scopes use the same convention.
g_i, G	Step implementation and complete program $G = g_1 \odot \cdots \odot g_m$.
$\mathcal{G}(P)$	Programs preserving the ordered subgoals and components of P.
$G_{j,h}^{(k,\mathrm{role})}$	Seed, parent, child, or current-plan incumbent, as indicated by the role tag.
$G_j^{(\mathrm{scope},\star)}$	Best current-island incumbent (global) or best retained program (archive); add h for local updates.
$H_{j,h}^{(k)}$	Recent-progress estimate controlling plan review.
$\mathcal{I}_{j,h}^{(k)}$	Inspiration programs actually sampled for an update.
K, J, W	Island count, global-round count, and local code visits after an accepted plan proposal.
M_{dbg}	Debugging-attempt limit for bounded generation calls.
$n_j^{(k)}, N_j$	Island and total code-improvement visit counts; retries do not create extra visits.
$P_j^{(k)}$	Plan associated with island k.
$Q_j(s)$	Mean component contribution since its last description change.
$R_{j,h}^{(k)}, r_{j,h}^{(k)}$	Discounted accumulated reward and current-visit reward.
$\mathcal{R}_j$	Existing components selected for review.
$s, \mathcal{S}_j, s_{j,i}^{(k)}$	Component, shared library, and component selected at step i.
$u_{j,i}^{(k)}, a_{j,i}^{(k)}$	Step subgoal and preliminary implementation annotation.
$V_{j,h}^{(k)}$	Discounted exposure statistic.
$\beta_{j,i}^{(k)}, \zeta_{j,i}^{(k)}$	Step time allocation and step type.
$\delta_{j,h}^{(k)}$	Nonnegative relative improvement in a code visit.
$\Delta_{j,h,i}^{(k)}, \mathbf{\Delta}_j(s)$	Step contribution and component history since its last description change.
$\Phi_{j,h}^{(k)}$	Interpreter report for a parent–child comparison.
$\perp$	No valid program obtained; not a numerical fitness.

C.2 Agent Description

Component Proposer. The Component Proposer receives the target problem description and proposes 10–20 relevant high-level components. The descriptions should cover distinct approaches

without duplicating components that differ only in implementation details. Each component should be specific enough to guide planning while supporting multiple concrete algorithms.

Planner. The Planner selects components from the library and organizes them into K different plans, with one plan assigned to each island. Each plan contains an ordered sequence of steps. The *subgoal and selected component* are fixed during ordinary code-level evolution. The *implementation annotation, step type, and time allocation* are revisable implementation fields. The annotation describes a preliminary procedure that the Coder can follow directly. This preserves the distinction between the plan's structural specification and its implementation guidance. The Planner distinguishes three step types, listed below. It assigns time allocations according to these execution requirements and the shared per-instance budget.

- **Complete-algorithm.** Runs a prescribed algorithm to completion. Its computational cost is described by an explicit time-complexity expression, which supports estimating the time needed for the step.
- **Budget-controlled.** Performs search within an allocated time limit. Its execution is controlled by a bounded duration rather than a prescribed completion-time expression.
- **Hybrid.** Combines a required complete-algorithm component with budget-controlled search. It requires a minimum time allocation for the mandatory component, with additional time available for search.

Coder. The Coder translates a complete plan into an executable program. It faithfully implements the preliminary procedures, preserves the ordered subgoals and selected components, and connects intermediate outputs so that later steps can use earlier results. It also exposes intermediate measurements for assessing each step. The Coder translates a supplied plan into an initial executable program. The initial-generation prompt requires all plan steps to be implemented in order, following their annotations, step types, and runtime fields. Intermediate outputs are passed between steps, and the program reports a solution and cumulative execution time after each step. The Code Improver handles subsequent implementation changes, runtime reallocation, and permitted step deactivation. The generated program reports each step's objective value or gap and execution time.

Debugger. The Debugger handles programs rejected because of compilation errors, invalid solutions, time-limit exceedances, or other execution failures. It supports two modes. In *direct repair*, the agent receives the rejected source code and its error message. It revises that implementation to address the reported failure while preserving the applicable plan requirements. In *error-aware regeneration*, the agent generates replacement code from the original generation context, augmented with the error message. Rather than requiring a modification of the rejected implementation, this mode warns the agent against repeating the same failure during a new generation attempt. Every repaired or regenerated candidate is evaluated again. During evolution, debugging is bounded by M_{dbg}. During initialization, seed generation continues until validation succeeds, and the main evolution procedure begins only after every island has a valid seed.

Code Improver. The Code Improver receives a parent program, inspiration programs, the current plan, the component library, and any available code-level revision suggestions. It generates a child while preserving the ordered subgoals and selected components. Its revisions may change concrete algorithms, data structures, parameters, implementation annotations, step types, stopping rules, time allocations, and interactions between adjacent steps. The parent and inspirations are selected by the system's *SelectCode* function. The Code Improver may also deactivate a step judged unnecessary, harmful, or too expensive by assigning it zero runtime while retaining its step identifier, component assignment, subgoal, and step type. The reason for deactivation is recorded in the step metadata. When a component has been removed from the active library, the Code Improver also receives guidance to bypass the corresponding step. The Plan Improver remains responsible for deleting the recorded plan entry or replacing its assigned component.

Plan Improver. The Plan Improver receives the selected island's current plan and programs, details of the best-performing plan, and sampled plans from other islands. These records include components, subgoals, implementation annotations, time allocations, and measured performance. The agent also receives the complete active component library and accumulated plan-level feedback. The proposed plan may add, remove, replace, or reorder steps and change their subgoals or selected

components. It includes preliminary implementation procedures and time allocations for the Coder. If the current plan references a removed component, the proposal must delete the affected step or replace its component with an available alternative. Every proposal that produces a valid seed starts a new plan generation, even when its content matches the preceding plan.

Component Improver. The Component Improver reviews selected components using their contribution histories and aggregated Interpreter feedback. It may refine descriptions, remove ineffective components, or consider missing high-level components suggested by the Interpreter. Only existing components used by the selected island's current plan are eligible for revision or removal during that round. Existing components absent from the plan remain unchanged. Suggestions for genuinely new components are considered separately. A description revision clears that component's numerical contribution history. A removal deletes its description from the active library but does not immediately rewrite existing programs or plans. Subsequent code improvement receives a suggestion to bypass the affected step, while subsequent plan improvement must delete or replace it.

Interpreter. The Interpreter compares the parent and child using their source code and implementation feedback. This feedback includes each step's objective value or gap, execution time, and relevant execution diagnostics. The actual optimization solutions, such as complete schedules or decision-variable assignments, are not included in the comparison context. The Interpreter explains measured outcomes rather than replacing the evaluator's assessment. Its report contains four components, detailed below. Code-level insights are associated with the parent and child. Plan-level reports accumulate for the current plan generation; reports from retired generations remain historical records rather than entering the new generation's active feedback pool. Component-level reports accumulate across relevant evaluations and islands. A proposed addition must describe a missing high-level search family rather than only an algorithm, neighborhood, parameter, or solver setting.

- **Change Summary:** Records the measured outcome as *better*, *worse*, or *equivalent*. Each material change identifies the affected step, change type, parent and child behaviors, observed effect, and attribution confidence. The summary becomes part of the child's evolution history.
- **Code-level Insight:** Assesses implementation changes under the unchanged subgoals and components. The newly generated code is identified as *effective*, *regression*, *neutral*, or *not established*. To support the argument, the interpreter includes supporting evidence and a concrete implementation direction. These insights are available to later Code Improver calls.
- **Plan-level Insight:** Assesses step usefulness, component suitability, organization, and runtime allocation. Step assessments use *helpful*, *supporting*, *implementation limited*, *component mismatch*, *subgoal unhelpful*, or *not established*. Structural recommendations use *revise*, *add*, or *remove*.
- **Component-level Insight:** Produces one record per unique component used by the plan, consolidating evidence when several steps share that component. It records observed implementations, transferable lessons, and proposed operations such as *keep*, *propose revise*, or *propose add*.

C.3 INITIALIZATION

Initialization constructs the component library, generates K plans, and obtains one validated seed for every island. It then performs W local code-improvement visits per island before global scheduling begins. Component descriptions remain unchanged during these initial local visits. Algorithm C1 gives the initialization procedure.

In Algorithm C1, *CodePlan* and *ImproveCode* include generation, evaluation, and debugging internally. Each successful call returns the validated program together with its average training fitness. Evaluation records remain available for subsequent interpretation.

During initialization, *CodePlan* continues repairing or regenerating a seed until validation succeeds. *ImproveCode* instead uses bounded debugging. If no valid child is obtained within that limit, it returns $(\perp, \perp)$. The rejected candidate is not inserted into the Program Database, and the local visit receives zero reward and zero relative improvement.

Each valid seed initializes its island's parent-eligible collection and incumbent. Before local code improvement, R, V, H, n are initialized to $0, 1, 0, 0$, respectively. Seed generation does not count as code improvement.

Algorithm C1 The Initialization Procedure of HeurEvo.

Require: Problem description, K islands, training set $\mathcal{D}$, per-instance budget B, local steps $W \geq 1$, and debugging limit M_{dbg}.

$\mathcal{S}_0 \leftarrow \mathsf{ProposeComponents}(\text{problem}), \quad \mathcal{A} \leftarrow \varnothing$
$\{P_0^{(k)}\}_{k=1}^{K} \leftarrow \mathsf{CreatePlans}(\mathcal{S}_0, K, B)$
Initialize an empty contribution array for each $s \in \mathcal{S}_0$
// Seed generation
for $k = 1, \ldots, K$ **do**
  $(G_0^{(k,\mathrm{seed})}, F_0^{(k,\mathrm{seed})}) \leftarrow \mathsf{CodePlan}(P_0^{(k)}, \mathcal{S}_0, B)$
  Insert the validated seed into $\mathcal{A}$ and associate it with island k
  $\mathcal{A}_{0,0}^{(k,\mathrm{par})} \leftarrow \{G_0^{(k,\mathrm{seed})}\}, \quad F_{0,0}^{(k,\star)} \leftarrow F_0^{(k,\mathrm{seed})}$
  $(R_{0,0}^{(k)}, V_{0,0}^{(k)}, H_{0,0}^{(k)}, n_{0,0}^{(k)}) \leftarrow (0, 1, 0, 0)$
end for
// Local code evolution
for $k = 1, \ldots, K$ **do**
  for $h = 1, \ldots, W$ **do**
    $(G_{0,h}^{(k,\mathrm{par})}, \mathcal{I}_{0,h}^{(k)}) \leftarrow \mathsf{SelectCode}(\mathcal{A}_{0,h-1}^{(k,\mathrm{par})}, \mathcal{A})$
    $(G_{0,h}^{(k,\mathrm{child})}, F_{0,h}^{(k,\mathrm{child})}) \leftarrow \mathsf{ImproveCode}(k, G_{0,h}^{(k,\mathrm{par})}, \mathcal{I}_{0,h}^{(k)})$
    if $G_{0,h}^{(k,\mathrm{child})} \neq \perp$ **then**
      $\Phi_{0,h}^{(k)} \leftarrow \mathsf{Interpret}(G_{0,h}^{(k,\mathrm{par})}, G_{0,h}^{(k,\mathrm{child})})$
      $\mathsf{UpdateState}(k, G_{0,h}^{(k,\mathrm{child})}, F_{0,h}^{(k,\mathrm{child})}, \Phi_{0,h}^{(k)})$ ▷ Eqs. C2–C3
    else
      Apply Eq. C3 with $r_{0,h}^{(k)} = \delta_{0,h}^{(k)} = 0$; add no program
      Retain the parent-eligible collection and incumbent for the next local update
    end if
  end for
  Record the final plan, programs, feedback, and statistics of island k at index 0
end for
$\mathcal{A}_0 \leftarrow \mathcal{A}, \quad N_0 \leftarrow \sum_{k=1}^{K} n_0^{(k)} = KW$
return K initialized islands, $\mathcal{S}_0$, and $\mathcal{A}_0$

Every scheduled *ImproveCode* call counts as one visit, whether it succeeds immediately, requires debugging, or remains invalid. The successful update or the failure update increments n exactly once. Consequently,

$$n_0^{(k)} = W, \qquad N_0 = KW.$$

The requirement $W \geq 1$ ensures that every island has a positive visit count before the first UCB selection. Initialization is separate from the J global scheduling rounds.

C.4 Evolution

Location of the deferred details. The exact scheduling rules and complete evolution algorithm deferred from Section 3.3 are collected here. References below to the "main-text" selection or update rules refer to Eqs. C1–C3 in this subsection. Initialization and its starting values remain in Appendix C.3; restart, timeout, and invalid-child handling are specified in the operator descriptions below.

UCB island-selection rule. At the beginning of global round j, the controller selects an island using its pre-round statistics:

$$k_j = \arg\max_k \left[\frac{R_{j-1}^{(k)}}{V_{j-1}^{(k)}} + C\sqrt{\frac{\log N_{j-1}}{n_{j-1}^{(k)}}} \right], \qquad N_{j-1} = \sum_{\ell=1}^{K} n_{j-1}^{(\ell)}. \tag{C1}$$

Here, R and V summarize discounted rewards and exposure. The count n records scheduled code-improvement visits to one island, and N totals these visits across islands. The coefficient C controls

exploration. The first term favors recent normalized progress; the second encourages exploration of under-visited islands. Counts include unsuccessful visits, not individual debugging attempts. Initialization with $W \geq 1$ ensures positive counts before the first selection.

Normalized reward and relative improvement. Let h index local code-improvement visits in round j, with $k = k_j$. For a validated child, let $F_{j,h}^{(k,\mathrm{child})}$ denote its average training fitness under budget B. Using the selected island's incumbent $F_{j,h-1}^{(k,\star)}$ and the current global incumbent $F_{j,h-1}^{(\mathrm{global},\star)}$ immediately before evaluation, define

$$\begin{aligned} r_{j,h}^{(k)} &= \mathrm{clip}\left(\frac{F_{j,h}^{(k,\mathrm{child})} - F_{j,h-1}^{(k,\star)}}{\max\left(F_{j,h-1}^{(\mathrm{global},\star)} - F_{j,h-1}^{(k,\star)}, 0\right) + \epsilon}, 0, 1 \right), \\ \delta_{j,h}^{(k)} &= \max\left(\frac{F_{j,h}^{(k,\mathrm{child})} - F_{j,h-1}^{(k,\star)}}{\left|F_{j,h-1}^{(k,\star)}\right| + \epsilon}, 0 \right). \end{aligned} \tag{C2}$$

Here, $\mathrm{clip}(z, 0, 1)$ truncates z to $[0, 1]$, and $\epsilon > 0$ prevents division by zero. The reward compares the child's gain over the island's incumbent with that island's gap to the current global incumbent, both measured before the update. For example, if these incumbents have fitness 80 and 100, respectively, a child with fitness 90 receives reward $10/(20 + \epsilon) \approx 0.5$. A child that does not improve on the island's incumbent receives zero reward. If the island already matches the global incumbent, the denominator is ϵ, so any gain of at least ϵ receives reward 1. In contrast, δ measures nonnegative relative improvement within the island. These comparisons precede incumbent replacement. The Interpreter instead compares the child with its sampled parent.

Discounted scheduling updates. Each scheduled code-improvement visit updates the selected island's statistics:

$$\begin{aligned} R_{j,h}^{(k)} &= \rho R_{j,h-1}^{(k)} + r_{j,h}^{(k)}, & V_{j,h}^{(k)} &= \rho V_{j,h-1}^{(k)} + 1, \\ H_{j,h}^{(k)} &= \rho H_{j,h-1}^{(k)} + (1-\rho)\delta_{j,h}^{(k)}, & n_{j,h}^{(k)} &= n_{j,h-1}^{(k)} + 1. \end{aligned} \tag{C3}$$

The factor $\rho \in [0, 1)$ controls discounting, and H is the recent-progress signal used for plan review. If no valid child is obtained, the same update is applied once with $r = \delta = 0$, without fabricating a fitness or replacing the incumbent. Non-selected islands retain their statistics.

Plateau criterion and complete procedure. For the selected island, $H_{j-1}^{(k)} \geq \tau$ results in one ordinary code-improvement visit, whereas $H_{j-1}^{(k)} < \tau$ triggers a plan review. A proposal starts a new plan generation only after its seed is validated, followed by W consecutive local visits without another island selection or threshold test. If seed generation fails, the old plan and programs remain active and the round proceeds with one ordinary code-improvement visit.

Algorithm C2 makes this update order explicit. As in Algorithm C1, *CodePlan* and *ImproveCode* include evaluation and debugging internally and return a validated program and its measured fitness, or $(\perp, \perp)$ after bounded failure during evolution. The temporary proposal $\widetilde{P}$ does not replace the current plan until *RestartIsland* succeeds.

Compact controller notation in the main text. Equations C.4 suppress round indices. Immediately before global round j, the compact variables correspond to

$$k^\star = k_j, \quad \bar{r}_k = \frac{R_{j-1}^{(k)}}{V_{j-1}^{(k)}}, \quad n_k = n_{j-1}^{(k)}, \quad n_{\mathrm{tot}} = N_{j-1}, \quad H_k = H_{j-1}^{(k)}.$$

They express the same UCB selection and plateau criterion specified above, without changing the reward or update rules.

Algorithm C2 The Evolution Procedure of HeurEvo.

Require: K initialized islands, training set $\mathcal{D}$, per-instance budget B, rounds J, local steps $W \geq 1$, threshold τ, library $\mathcal{S}_0$, codebase archive $\mathcal{A}_0$, and debugging limit M_{dbg}.

for $j = 1, \ldots, J$ **do**
  Carry forward all islands' plans, populations, feedback, and statistics
  $k \leftarrow \mathsf{SelectIsland}(j), \quad L \leftarrow 1$ ▷ Eq. C1
  Initialize local state $h = 0$ from the preceding round
  if $H_{j-1}^{(k)} < \tau$ **then**
    $\widetilde{P} \leftarrow \mathsf{ImprovePlan}(k)$
    $(G_j^{(k,\mathrm{seed})}, F_j^{(k,\mathrm{seed})}) \leftarrow \mathsf{CodePlan}(\widetilde{P}, \mathcal{S}_{j-1}, B)$
    if $G_j^{(k,\mathrm{seed})} \neq \perp$ **then**
      $\mathsf{RestartIsland}(k, \widetilde{P}, G_j^{(k,\mathrm{seed})}, F_j^{(k,\mathrm{seed})})$
      $L \leftarrow W$
    end if
  end if
  for $h = 1, \ldots, L$ **do**
    Refresh the global incumbent from the current island incumbents
    $(G_{j,h}^{(k,\mathrm{par})}, \mathcal{I}_{j,h}^{(k)}) \leftarrow \mathsf{SelectCode}(\mathcal{C}_{j,h-1}^{(k)}, \mathcal{A}_{j,h-1})$
    $(G_{j,h}^{(k,\mathrm{child})}, F_{j,h}^{(k,\mathrm{child})}) \leftarrow \mathsf{ImproveCode}(k, G_{j,h}^{(k,\mathrm{par})}, \mathcal{I}_{j,h}^{(k)})$
    if $G_{j,h}^{(k,\mathrm{child})} \neq \perp$ **then**
      $\Phi_{j,h}^{(k)} \leftarrow \mathsf{Interpret}(G_{j,h}^{(k,\mathrm{par})}, G_{j,h}^{(k,\mathrm{child})})$
      $\mathsf{UpdateState}(k, G_{j,h}^{(k,\mathrm{child})}, F_{j,h}^{(k,\mathrm{child})}, \Phi_{j,h}^{(k)})$ ▷ Eqs. C2–C3
    else
      Apply Eq. C3 with $r_{j,h}^{(k)} = \delta_{j,h}^{(k)} = 0$
      Retain programs, incumbent, and feedback; add no program
    end if
  end for
  Record the selected island's final local state and archive at round j
  $N_j \leftarrow \sum_{\ell=1}^{K} n_j^{(\ell)}$
  $\mathcal{R}_j \leftarrow \mathsf{SelectComponents}(\mathcal{S}_{j-1})$ ▷ Eq. C8
  $\mathcal{S}_j \leftarrow \mathsf{ImproveComponents}(\mathcal{S}_{j-1}, \mathcal{R}_j)$
end for
return $\arg\max_{G \in \mathcal{A}_J} F(G)$

Table C2: System functions and agent-response calls.

Call category	Operators
System functions	*SelectIsland*, *SelectCode*, *RestartIsland*, *UpdateState*, *SelectComponents*
Agent-response calls	*ImprovePlan*, *CodePlan*, *ImproveCode*, *Interpret*, *ImproveComponents*

System functions and agent-response calls. System functions execute prescribed sampling, database, evaluation, and numerical-update rules. Agent-response calls construct prompts, obtain structured responses, and parse the requested outputs.

CodePlan and *ImproveCode* additionally include system-controlled evaluation and debugging. Their agents propose source code, but the returned fitness is measured by execution rather than supplied by an agent. The internal evaluator also determines whether a candidate must be sent to the Debugger. No separate *EvaluateAndDebug* call is required in the pseudocode.

Island selection (*SelectIsland*). *SelectIsland* computes the UCB scores using Eq. C1 in the main text and returns the selected island index. The decision uses the statistics available before the current round.

For an ordinary code-improvement visit, local index $h = 0$ refers to the programs, incumbents, and statistics carried from the preceding global round. After an accepted plan proposal, $h = 0$ instead refers to the new seed and the quantities initialized by *RestartIsland*.

The cumulative counts n and N retain their history across plan generations. They are distinct from V, the discounted exposure statistic that is reset when a new generation starts.

Plan proposal (*ImprovePlan*). *ImprovePlan* gathers the selected island's plan and programs, cross-island plan details, active component descriptions, and accumulated plan-level feedback. It passes them to the Plan Improver and parses the returned proposal.

The existing plan remains available while the proposal is being implemented and validated. A proposal does not replace the current plan before a valid seed is obtained. Plan equality is not an acceptance criterion.

Seed generation and validation (*CodePlan*). *CodePlan* passes the proposed plan, component descriptions, and execution budget to the Coder. It evaluates the generated program and invokes the Debugger when compilation, validity, or runtime checks fail.

A successful call returns

$$\left(G^{(\mathrm{seed})}, F^{(\mathrm{seed})}\right),$$

where the program is the final validated implementation and the fitness comes from its evaluation.

Initialization continues this process until a valid seed is obtained. During plan evolution, the configured debugging limit applies. If debugging is exhausted, *CodePlan* returns $(\perp, \perp)$. The proposal is then discarded, the island retains its previous plan and programs, and the round resumes with one ordinary code-improvement visit.

A failed seed-generation call does not increment the code-improvement visit count and does not itself trigger a scheduling-reward update.

Island restart (*RestartIsland*). *RestartIsland* is invoked only after a proposed plan produces a valid seed. It installs the proposed plan and initializes its parent-eligible collection and incumbent with that seed. This handling applies even when the proposal has identical content to the previous plan or its seed has lower fitness than the previous incumbent.

Programs from the previous generation remain in the codebase according to the database's retention policy, but they are no longer eligible parents for the new generation. Previous plan-level feedback remains archived, and a new current-generation feedback pool is started.

The function resets R, V, H to $0, 1, 0$, respectively, without resetting n or N. The seed establishes a new within-generation baseline and does not receive an improvement reward against the retired incumbent.

The island then receives W consecutive local code-improvement visits. Neither island selection nor the plan-evolution threshold is reconsidered between these visits.

Parent and inspiration selection (*SelectCode*). *SelectCode* samples one parent from the selected island's current parent-eligible collection and inspiration programs from the codebase. The parent must belong to the current plan generation, while inspirations may provide examples from other plans.

The input $\mathcal{A}$ identifies the complete codebase. The collection $\mathcal{A}^{(k,\mathrm{ins})}$ identifies the programs available as inspirations for the selected island under the configured database policy. The returned $\mathcal{I}_{j,h}^{(k)}$ contains the programs actually sampled for that visit.

The reference backends use related but distinct mechanisms. OpenEvolve's documented parent-selection mechanisms combine random island sampling, elite selection, and fitness-weighted sampling. Its configuration supports both high-performing and diverse programs as prompt examples (Sharma, 2025). AdaEvolve's published sampling subroutine switches between uniform parent selection with diverse inspirations and parent selection from the highest-fitness quartile with high-fitness inspirations (Cemri et al., 2026).

HeurEvo applies its current-plan eligibility restriction through this database interface. Parent and inspiration selection are repeated before every local update, so later visits can use validated programs produced earlier in the same round.

Code generation and validation (*ImproveCode*). *ImproveCode* constructs the Code Improver's prompt from the sampled programs, current plan, component library, and available code-level feedback. It then evaluates the generated child and performs bounded debugging when necessary.

A successful call returns the final validated child and its measured average training fitness. If debugging produces a valid repaired or regenerated program, that final program—not an earlier rejected version—is returned.

When no valid child is obtained, the call returns $(\perp, \perp)$. Intermediate rejected versions are not inserted into the Program Database. All internal generation and debugging attempts remain part of the same code-improvement visit.

A valid child with degraded fitness is different from a rejected child. It can still be interpreted and submitted to the database's ordinary admission and retention procedure.

Program comparison (*Interpret*). *Interpret* retrieves the selected parent's evaluation record and the final validated child's record. It supplies the source code and per-step implementation feedback to the Interpreter and parses the four-part report.

When debugging has modified the child, the comparison uses the validated result and the originally selected parent. No additional program evaluation is required merely to construct this comparison.

An unrepaired child does not receive a normal parent–child performance interpretation because no valid child fitness is available. Its diagnostics have already been used by the Debugger.

Database insertion and updates (*UpdateState*). *UpdateState* computes the normalized reward and relative improvement for a valid child using the incumbent fitnesses immediately before the child's evaluation. It then applies Eqs. C2–C3 in the main text. Incumbent replacement occurs after these comparisons.

The Interpreter and scheduler use different references. The Interpreter compares the child with its sampled parent. Scheduling measures progress against the selected island's incumbent, which need not be the sampled parent.

The validated child is submitted to the Program Database with its measured fitness, island and plan-generation association, implementation measurements, and Interpreter report. The database manages retention and parent eligibility. Feedback is made available to subsequent calls, and measured component contributions are appended to the appropriate arrays.

If *ImproveCode* returns failure, the visit receives

$$r_{j,h}^{(k)} = 0, \qquad \delta_{j,h}^{(k)} = 0.$$

The main-text update rule is applied with these values, including one increment of n. No numerical child fitness is fabricated, no program is inserted, and the incumbent remains unchanged. The discounting rule reduces previously accumulated positive reward when no new reward is obtained.

A failed visit inside a W-visit block does not terminate the block. The procedure continues with the next local visit using the available parent-eligible programs. Rejected attempts do not automatically contribute zero-valued component observations, because scheduling rewards and measured step contributions serve different purposes.

Component selection (*SelectComponents*). *SelectComponents* considers existing components that are both present in the active library and used by the selected island's current plan. A component also needs at least one contribution observation collected under its current description.

The function computes the review probabilities defined in Appendix C.5 and returns $\mathcal{R}_j$. Existing components absent from the selected plan remain unchanged during that round. New-component suggestions are considered separately from the contribution-based selection of existing components.

Component revision and deferred removal (*ImproveComponents*). *ImproveComponents* passes the selected components, contribution histories, aggregated feedback, and addition suggestions to the Component Improver. Selection requests a review rather than requiring an edit.

When a description is revised, only that component's contribution array is cleared. Arrays for unchanged components are retained. New components begin with empty arrays.

When a component is removed, its description is deleted from the active library. Existing plans and stored programs are not immediately rewritten, and their measured fitness values remain associated with the original programs. Subsequent code-improvement prompts suggest bypassing the affected step with zero allocated time. Any child implementing that suggestion must pass ordinary validation. Deletion or replacement of the recorded plan entry occurs when the Plan Improver next revises the affected plan. It must delete that entry or replace its component with an available alternative.

Completion of a global round. Local visits are processed sequentially. Each visit uses the latest parent-eligible collection, codebase, incumbent, and feedback. The global reference is refreshed from the current island incumbents before each child evaluation.

After the final local visit, the selected island's resulting programs, feedback, and statistics are recorded for round j. Non-selected islands retain their corresponding quantities. Component review then produces the library used by subsequent rounds.

The current global incumbent is the best program among the current island plans. The archive-best program is the best retained program in the codebase, which may belong to a retired generation. Algorithm C2 returns the latter.

C.5 Component Contribution and Review

Intermediate fitness. For a program with m steps, let y_i denote its average best-so-far training fitness after step i, with final fitness $y = y_m$. The recorded sequence satisfies

$$y_1 \le y_2 \le \cdots \le y_m = y.$$

These are higher-is-better fitness values, rather than raw minimization objectives or gaps. Equality is allowed when a step does not improve the incumbent.

Let y_g denote the current global incumbent immediately before evaluation. The first-step reference is the mean of the recorded first-step observations available to the contribution calculation,

$$\bar{y}_1 = \frac{1}{M_{\text{first}}} \sum_{q=1}^{M_{\text{first}}} y_{q,1}. \tag{C4}$$

Here, M_{first} counts first-step observations and is distinct from the code-improvement visit count N.

Contributions in multi-step programs. For $m \ge 2$, the observed contribution of step i is

$$\Delta_i = \begin{cases} 0, & i = 1,\ y - \bar{y}_1 \le 0, \\ \operatorname{clip}\left(\dfrac{y_1 - \bar{y}_1}{y - \bar{y}_1 + \epsilon}, 0, 1 \right), & i = 1,\ y - \bar{y}_1 > 0, \\ \operatorname{clip}\left(\dfrac{y_i - y_{i-1}}{y - y_{i-1} + \epsilon}, 0, 1 \right), & 2 \le i < m, \\ \operatorname{clip}\left(\dfrac{y_i - y_{i-1}}{\max(y_g, y) - y_{i-1} + \epsilon}, 0, 1 \right), & i = m. \end{cases} \tag{C5}$$

The first step is compared with the historical first-step reference. Intermediate steps are normalized by the remaining improvement obtained by the complete program. The final step is normalized by the remaining distance to the better of the program's final fitness and the pre-evaluation global incumbent. The score prioritizes component review rather than estimating a causal effect.

With full indices restored, the contribution of step i in local update h is denoted by $\Delta^{(k)}_{j,h,i}$.

Contributions in single-step programs. For $m = 1$, the only step is also the final step. Setting $y_0 = \bar{y}_1$, its contribution is

$$\Delta_1 = \begin{cases} 0, & \max(y_g, y_1) \le \bar{y}_1, \\ \mathrm{clip}\left(\dfrac{y_1 - \bar{y}_1}{\max(y_g, y_1) - \bar{y}_1 + \epsilon}, 0, 1\right), & \max(y_g, y_1) > \bar{y}_1. \end{cases} \tag{C6}$$

The special case assigns zero contribution when the historical reference is at least as large as both the program's outcome and the current global incumbent. Otherwise, the denominator is strictly positive.

Per-component contribution histories. Each component maintains an independent contribution array,

$$\boldsymbol{\Delta}_j(s) = \left[\Delta_s^{[1]}, \ldots, \Delta_s^{[M_j(s)]}\right].$$

The array contains observations collected since that component's most recent description change. Each measured occurrence contributes one observation. A component used in several steps can therefore receive several numerical observations from one program, while its qualitative Interpreter feedback is consolidated into one component-level record.

For a nonempty array,

$$Q_j(s) = \frac{1}{M_j(s)} \sum_{q=1}^{M_j(s)} \Delta_s^{[q]}. \tag{C7}$$

A description revision clears the corresponding array. Observations associated with the previous description do not enter the revised component's numerical average. Historical Interpreter reports can remain available as contextual evidence. New components begin with empty arrays, and unchanged components retain their existing observations.

Review eligibility and probability. An existing component is eligible for review when it remains in the active library, appears in the selected island's current plan, and has at least one contribution observation under its current description. Thus,

$$\mathcal{R}_j \subseteq \left\{ s \in \mathcal{S}_{j-1} : \ s \text{ is used by } P_j^{(k_j)},\ M_j(s) \ge 1 \right\}.$$

For an eligible component, the review probability is

$$p_j(s) = I_{\min} + \frac{I_{\max} - I_{\min}}{1 + C_I \sqrt{Q_j(s)}}, \tag{C8}$$

where $0 \le I_{\min} \le I_{\max} \le 1$ and $C_I > 0$. A lower contribution score increases the probability of review without making revision mandatory.

There is no requirement to accumulate W observations before reviewing a component. The W consecutive code-improvement visits after an accepted plan proposal provide opportunities to gather multiple observations under unchanged descriptions, but they do not impose a minimum history length. After a description revision, one new measured contribution is sufficient for eligibility.

Unrepaired visits do not produce successful step measurements and therefore do not guarantee additional contribution observations. Suggestions for adding new components are assessed separately from the numerical review of existing components.

D PROMPT TEMPLATES

This appendix presents the prompt templates for initial solver generation and plan-aware code evolution, together with their shared task context and solution-reporting requirements. Fields in braces are filled with task-specific information, optimization plans, and evaluation feedback.

D.1 TASK CONTEXT AND SUBSTITUTION FIELDS

The shared `problem_context` contains the natural-language problem description, a runtime command and parameter values, CSV table names and locations, column descriptions, the mathematical model, and the reference Gurobi solver code. The reference implementation is authoritative when it disagrees with a textual description. The first selected training instance supplies the representative prompt context. The resulting program is evaluated on the selected training instances. The table descriptions identify the data that the program must read at execution time; CSV rows are not embedded in the prompt. Runtime parameter values must be parsed by the generated program and can vary across executions.

Fields in braces are replaced when a message is constructed. Their meanings are:

- `problem_context`: the shared task context described above.
- `time_limit`: the configured per-instance runtime allowance in seconds.
- `formatted_plan` and `strategy_context`: the validated runtime-aware plan and the descriptions of the strategy families selected by that plan. In the evolution prompt, `plan` provides the island's fixed strategy and subgoal for each step, and `selected_strategies` supplies the corresponding strategy descriptions.
- `solution_output_requirements`: the task-specific contract for a complete solution report that can be checked by the verifier. `intermediate_output_requirements` inserts the shared reporting contract reproduced in Section D.2.
- `metrics`, `fitness_description`, `improvement_areas`, and `artifacts`: the parent program's recorded performance, the scoring rule, comparisons with previous attempts, and execution feedback.
- `previous_attempts`, `other_context_programs`, and `search_guidance`: the selected ancestor history, top-performing and inspiration programs, and available guidance from prior evaluations.
- `current_program` and `language`: the parent source code and its language. `timeout_warning` reiterates the configured runtime allowance and the requirement to preserve the program's output format.

Instance-specific models, program histories, plans, and output requirements are represented by placeholders so that the templates remain reusable across tasks.

D.2 SHARED INTERMEDIATE-REPORT CONTRACT

Both user prompts require a complete solution report after each plan step and at termination. The following instructions specify the shared reporting format and cumulative elapsed-time measurements.

Shared intermediate-report contract

```
The program must print one complete solution report after every plan
step, not only at the end. The body of every `<stepK>` block and the
`<final>` block must independently follow the complete solution report
contract above.

After a step that does not improve the incumbent, repeat the complete
report for the current best incumbent. If no incumbent exists, print
the complete no-feasible-solution report permitted by the contract.

Use this stdout structure:
```

Shared intermediate-report contract (continued)

```
<step1>
# One complete solution report after the first plan step.
</step1>
Cumulative output time: <elapsed seconds> seconds

<step2>
# One complete solution report after the second plan step.
</step2>
Cumulative output time: <elapsed seconds> seconds

...

<stepN>
# One complete solution report after the N-th plan step.
</stepN>
Cumulative output time: <elapsed seconds> seconds

<final>
# One complete final solution report.
</final>

Rules:
1. Print one `<stepK>...</stepK>` block for each plan step, in plan
order.
2. Number step tags consecutively by plan step output: `<step1>`,
`<step2>`, ..., `<stepN>`.
3. Print each step report only after that step's required operation
has run. Never print a partial solution, debug text, traceback,
summary, or "same as above" inside a report block.
4. Print `Cumulative output time: <elapsed seconds> seconds`
immediately after each `</stepK>` tag, using wall-clock time since
program start. This timer is for reporting only and must not affect
control flow.
5. The `<final>` block is mandatory. Even when it duplicates
`<stepN>`, repeat the complete solution report.
```

D.3 Initial program generation

Unlike subsequent code evolution, initial program generation follows the supplied annotations and runtime fields and is instructed to implement every plan step. The system message defines the agent's role, and the user message supplies the task, plan, and implementation requirements.

Initial-generation system message

```
You are an expert optimization coding agent. Your task is to
translate a provided optimization plan into a complete, executable
Python program.
```

Initial-generation user-message template

```
{problem_context}

# Task

Solve the optimization problem above by generating one complete,
standalone Python program that faithfully implements every step of
the supplied optimization plan and returns the best feasible solution
found.
```

Initial-generation user-message template (continued)

```
The supplied plan is the algorithmic specification for solving this
problem. Every plan step is mandatory, and each `Annotation` describes
an operation that must appear in the executable implementation. Do
not replace a required strategy or operation merely because another
method is easier to implement or faster to run.

# Solver Policy

Gurobi (`gurobipy`) is the only permitted mathematical optimization
backend. If a plan step uses a solver, optimizer,
mathematical-programming backend, or optimization subproblem,
implement it with `gurobipy`. Do not import or call another optimizer.

Standard Python and non-optimizer libraries may be used for
deterministic computation, data handling, numerical operations, and
validation. Using Gurobi as the backend does not permit replacing the
algorithm, formulation, transformation, neighborhood, repair
mechanism, or other operation required by the plan.

{strategy_context}

# Step-by-Step Plan to Implement

{formatted_plan}

# Runtime-Aware Plan-Following Contract

The Planner has already classified and scheduled every step for a
per-instance runtime budget of {time_limit} seconds, and that
schedule has been validated. It is not yours to revisit: do not
classify a step again, choose or revise an estimate, reallocate
runtime between steps, or write a new Reason. Your task is to
implement what each `Annotation` describes, under the runtime
semantics the plan already declares.

The step-by-step plan above is the implementation-facing rendering of
a validated strict JSON plan. Every field it shows -- `Strategy`,
`Subgoal`, `Annotation`, `Step type`, `Estimated running time`, and
`Reason` -- is authoritative as written.

Implement the plan so that its declared runtime semantics hold:

1. For a Complete-algorithm step, execute the finite operation
completely to its scale-derived stopping rule, without an
elapsed-time cutoff.
2. For a Budget-controlled step, run one shared local budget `B` for
the whole step, equal to that step's `Estimated running time`. `B` is
a floor as well as a ceiling: the step must not overrun it and must
not finish before it has spent `B` on useful work of its own Strategy.
Repeated bounded calls and nested loops within the step share that
same `B`; they do not receive separate budgets. When one pass
converges before `B` is spent, use the remaining budget for further
work of the same Strategy -- for example, another restart from a
different starting point, a perturbation of the incumbent, a wider
neighborhood, or a finer parameter -- and keep the best feasible
result found. A step whose `Estimated running time` is `remaining
time` takes its `B` from the remaining-allowance rule below instead.
```

Initial-generation user-message template (continued)

```
3. For a Hybrid step, execute the complete `C` portion unconditionally
and then run one shared local budget `B` for the bounded portion only.
The bounded portion follows the same floor-and-ceiling and
useful-work requirements as a Budget-controlled step: it must spend
all of `B` on work of its own Strategy without overrunning it.
`Reason` states how the step's `Estimated running time` splits into `C
+ B`.

Time counts toward `B` only when the step is performing useful work
required by its own `Annotation` and Strategy. Sleeping, busy-waiting,
repeatedly printing, repeatedly validating an unchanged incumbent
solely to consume time, executing an empty or unconditional-break
loop, or repeating work that cannot produce a new candidate,
decision, or required artifact does not count as spending `B`.

The plan's last step may state `remaining time` in place of a number.
It is the one step whose budget the plan does not fix: record one
shared program start with a monotonic clock at the beginning of the
timed execution, and set that step's `B` to {time_limit} seconds minus
the elapsed time since that start, clamped at zero. Every part of the
step, including its own setup, model construction, and result
extraction, is charged to that same `B`, and enough of `B` is reserved
for the step's required report. A solver time limit set inside this
step bounds the solve alone, so derive it from the budget still
remaining at the moment of the call -- after model construction --
minus the reserve that extraction and the report will need. Its
bounded improvement work is permitted bounded work rather than an
unconditional core: when no usable time remains, skip the improvement
and still produce the step's required report. This shared start time
measures that step's budget only; it is not a deadline over any other
step.

A local time check may control only budget-controlled work. It must
never skip a complete portion, an entire mandatory step, or any later
plan step. The generated source must not contain or enforce the total
per-instance budget or any program-wide deadline other than the
remaining-time step's own `B` described above, and must not sum,
rescale, or otherwise recompute the step estimates.

# Annotation Fidelity Requirements

Before coding, privately split every `Annotation` into its concrete
required claims and map each claim to executable statements. Only
clauses explicitly marked optional may be omitted.

- Every required artifact must materially affect a later model,
candidate choice, solver setting, acceptance decision, repair,
validation, or required operation. Computing it and then ignoring it
is not implementation.
- Implement the defining mechanics of every named algorithm,
formulation, transformation, derivative rule, stopping rule, and
conditional rule. Do not substitute a proxy or generic alternative
unless the `Annotation` permits it.
- Prepare valid fallback input before each mandatory operation so
that a missing incumbent, empty preferred input, or failed bounded
call cannot turn the step into a no-op.
- Handle expected failures explicitly. Do not hide mandatory work
behind broad exception handling.
- Each step's required operation runs after the preceding step's
report and before its own `<stepK>` report. Two `<stepK>` reports with
no computation between them mean the later step was never
implemented.
```

Initial-generation user-message template (continued)

```
- When one step produces guidance for another operation, pass the
exact artifact forward and make it affect the consumer's behavior.
- Use only solver-variable and constraint keys that were actually
created, with an explicit missing-value policy.

# Program Implementation Requirements

The code must:

1. Read all required CSV files.
2. Implement every plan step in order. A step is implemented only
when every non-optional claim in its `Annotation` is realized by
executable operations during that step's own execution interval. The
operations must run even if they find no improvement. Failure to
improve the incumbent is permitted; failure to perform the required
search, construction, transformation, repair, validation, or solver
operation is not.
3. Maintain the best feasible incumbent found so far and pass it to
later steps when applicable.
4. Execute every step unconditionally. Every Budget-controlled step
must spend its full `B`, and every Hybrid step must execute its
complete `C` portion and then spend its full bounded `B`. A step-local
bound governs how long that step's work runs; it does not license the
step to finish early. When a pass converges with budget left, spend
the remainder on further work of the same Strategy -- another
restart, a perturbation of the incumbent, a wider neighborhood -- and
keep the best result. A step-local bound may limit improvement
intensity but may not skip the step's mandatory core or a later step.
5. Use strategy-faithful fallback construction, repair, and
validation when preferred input or a solver result is unavailable.
6. Express every mathematical optimization relationship and
subproblem through `gurobipy`.
7. Make every preprocessing or guidance artifact affect the operation
specified by its `Annotation`.
8. Follow the complete-solution-report contract exactly after every
step and for the final report.

## Code comment requirements

In the generated Python code, clearly mark the plan structure using
Python comments only. These comments are metadata for later agents;
they must not be printed to stdout.

For every plan step, include exactly one plan comment block, in plan
order. Every field in the block is copied from the plan: the step
number, `Strategy`, `Subgoal`, `Annotation`, `Step type`, `Estimated
running time`, and `Reason`. Reproduce each value exactly as the plan
states it. Do not rename, summarize, rephrase, reorder, or omit a
field; do not add punctuation that the plan does not already have;
and do not write a value of your own for any of them. The executable
implementation itself must faithfully realize the copied `Annotation`;
do not add a separate `Approach` field.

Every plan step is mandatory. Implement every step with substantive
executable logic. Execute every step unconditionally and in plan
order. Never skip, omit, disable, defer, or replace a step, including
when preferred input is empty, a solver call fails, or a step's
budget is small. A step-local bound may control only that step's
permitted bounded work; it must not prevent the step's mandatory core
or any later step from executing and producing its report.

For every implemented step, include comments in this form before the
relevant code, copying every value from the plan:
```

Initial-generation user-message template (continued)

```
# --------------------
# Step <number>:
# Strategy: <Strategy>
# Subgoal: <Subgoal>
# Annotation: <Annotation>
# Step type: <Step type>
# Estimated running time: <Estimated running time>
# Reason: <Reason>
# --------------------

Never combine, merge, or jointly execute two plan steps. Each step
must have its own execution interval, beginning only after the
preceding step's report and ending before its own report. Work
performed before that interval cannot be credited to the step.

Immediately before returning, privately audit every copied
`Annotation` against the executable code and correct any clause
represented only by a comment, unused computation, proxy strategy,
conditional no-op path, or optional call.

# Code Quality Requirements

The script must run without manual edits and must not hard-code the
final answer. Use reasonable numerical tolerances for feasibility
checks and for extracting integer or binary values.

Use only standard Python and commonly available non-optimizer
libraries. `gurobipy` must be the sole mathematical optimization
backend.

Before returning, privately verify that every plan step has exactly
one comment block whose seven fields appear in order and match the
plan exactly, every helper's return shape matches all call sites, and
no mandatory operation can be lost through a missing key, empty
input, failed solver call, or broad exception handler. Do not add
runtime code that reads its own source or validates its own comment
metadata.

# Program Output Requirements

The following contract defines one complete solution report:

<solution_output_requirements>
{solution_output_requirements}
</solution_output_requirements>

{intermediate_output_requirements}

# Response Format

Return exactly one `<code>...</code>` block. Do not include Markdown
fences or any text outside this block.

<code>
# Complete executable Python code only. Do not use Markdown fences
here.
</code>
```

D.4 PLAN-AWARE CODE EVOLUTION

The code-evolution prompt requests targeted SEARCH/REPLACE edits under a fixed recorded plan. It preserves step identifiers, component assignments, and subgoals while allowing implementation

changes, runtime reallocation, and step deactivation through zero-time allocation. The experimental template is reproduced below with its original field names. The edits must preserve the plan's strategies and subgoals, respect the runtime allowance, and retain the required solution-reporting format.

Code-evolution system message

```
You are an expert optimization engineer tasked with iteratively
improving a codebase. Your job is to analyze the current program and
suggest improvements based on feedback from previous attempts. Focus
on making targeted changes that will increase the program's
performance metrics.
```

Plan-aware code-evolution user-message template

```
{problem_context}

# Program Output Requirements

The following contract defines one complete solution report:

<solution_output_requirements>
{solution_output_requirements}
</solution_output_requirements>

{intermediate_output_requirements}

# Current Program Information
- Current performance metrics: {metrics}
- Areas identified for improvement:
    - {improvement_areas}
    - {fitness_description}
- Plan implemented by the current program:
{plan}

## Last Execution Output
{artifacts}

# Program Evolution History
## Previous Attempts

{previous_attempts}

{other_context_programs}

# Current Program
```{language}
{current_program}
```

{search_guidance}

Available Problem-specific Strategies

{selected_strategies}

Task

Improve the Current Program so that it scores better on the metric
named under "# Current Program Information", by finding a better
implementation of the plan it already carries. Every constraint below
is binding.
```

Plan-aware code-evolution user-message template (continued)

```
Answer with SEARCH/REPLACE diffs and nothing else:

<<<<<<< SEARCH
# Original code to find and replace (must match exactly)
=======
# New replacement code
>>>>>>> REPLACE

Example of valid diff format:
<<<<<<< SEARCH
for i in range(m):
    for j in range(p):
        for k in range(n):
            C[i, j] += A[i, k] * B[k, j]
=======
# Reorder loops for better memory access pattern
for i in range(m):
    for k in range(n):
        for j in range(p):
            C[i, j] += A[i, k] * B[k, j]
>>>>>>> REPLACE

Rules for the answer itself:
- Send as many blocks as you need. Every SEARCH section must match
code in "# Current Program" character-for-character, including
whitespace and indentation. Do not paraphrase or reformat it.
- Explain your reasoning for each change.
- Give every function a concise docstring naming the approach it
implements.
- Do not write a Changes Description. This workflow derives one by
comparing the step comments before and after.

# Constraints

## The plan is fixed; the implementation is not

Each plan step appears in the code as one comment block, with exactly
these fields in this order:

# --------------------
# Step <number>:
# Strategy: <Strategy>
# Subgoal: <Subgoal>
# Annotation: <Annotation>
# Step type: <Complete-algorithm step | Budget-controlled step |
Hybrid step>
# Estimated running time: <number seconds | remaining time>
# Reason: <basis for the step type and the number>
# --------------------

| Field | May you change it? |
| --- | --- |
| Step, Strategy, Subgoal | **No.** Reproduce them byte-for-byte. Do
not rename, rephrase, summarize, shorten or expand them. |
| Annotation | **Yes, and you must** whenever the implementation
changes. |
| Step type, Estimated running time, Reason | **Yes**, following
"Runtime budget" below. |

You CANNOT add, delete, renumber, reorder or merge steps.
```

Plan-aware code-evolution user-message template (continued)

```
The Annotation states the main operation, the variables, constraints
or structures it touches, and why that operation serves the fixed
Strategy and Subgoal. It must describe the code you are returning,
never the code you replaced.

Before returning, confirm that every step of the Current Program is
still present in the same order, that Step, Strategy and Subgoal are
untouched, and that every Annotation matches the code you actually
wrote.

## What you are free to change

Anything inside a step, as long as it stays inside the same strategy
family: a materially different mechanism from that family, compatible
tactics combined, better data structures or solver formulations, and
tuning such as neighborhood size, candidate ordering, iteration
counts, restart intensity, stopping rules and solver settings.

The strategy descriptions above are a design space, not text to copy.
Use them to find another way to implement a fixed Strategy for its
fixed Subgoal. Never relabel a step's Strategy or change its Subgoal
because some other family appears elsewhere in this prompt, and never
copy a description verbatim into plan metadata.

To drop a step you judge unnecessary, harmful or too expensive, keep
its Step, Strategy, Subgoal and Step type unchanged, and set
`Estimated running time: 0 seconds`. This is the one case where the
estimate is not the `C`, `B` or `C + B` that its step type would
otherwise imply. Its `Reason` still opens with the exact step type,
and then says the step was dropped and why.

## Runtime budget

`Estimated running time` is the time a step is expected to occupy in
the next run. What that number is depends on the step type, and only
one of the three is yours to choose:

1. **Complete-algorithm step.** The work has a finite, scale-derived
stopping rule and runs to completion without an elapsed-time cutoff.
The number is its natural runtime `C`, which you observe or predict
but do not pick.
2. **Budget-controlled step.** The work is bounded by a solver limit,
loop deadline, iteration budget, or equivalent local bound. The
number is the one shared local budget `B` you enforce in the code for
the whole step, including its repeated calls and nested loops. This
is the number you allocate.
3. **Hybrid step.** An unconditional complete portion plus bounded
work. The number is `C + B`, with one shared `B` covering the bounded
portion only.

A last step should be a `Budget-controlled step` and should read
`remaining time` in place of a number for "Estimated running time".
That is not an estimate you derive: it means the step takes the
remaining runtime allowance, the total minus actual elapsed time
since a shared program start, clamped at zero. Keep the value, and
keep the code that computes it, while that step still ends the
program in useful bounded improvement. Replace it with a number only
by also replacing that computation with one shared local budget `B`
you enforce yourself, and say so in its `Reason`.

### First, measure
```

Plan-aware code-evolution user-message template (continued)

```
"## Last Execution Output" reports each step's `Cumulative Time`:
wall-clock seconds since program start, averaged over the evaluation
instances. It is cumulative, so it is not the cost of that one step
-- a step's own measured runtime is its cumulative time minus the
previous step's, taking zero before the first step. Derive every
step's measured runtime that way before changing any number. Where a
run was cut short and a step never reported a cumulative time, it has
no measurement: say so in its `Reason` and estimate it the way a
reimplemented step is estimated below.

The numbers currently in the code are predictions made before
anything ran, and are often far from what the program actually did.
Prefer a measurement to an inherited number wherever a measurement
exists.

### Then decide each step

- **A step you leave as it is.** Its measured runtime is its cost, so
use that number.
- **A step whose measurement contradicts its type.** A
budget-controlled or hybrid step that finished well inside its budget
was stopped by its own rule rather than by the bound, which is what a
complete portion does. Correct the `Step type` -- to a
complete-algorithm step, or to a hybrid one with a smaller bounded
portion -- and let the number follow. Do not keep a type whose `B` the
code never actually reached. A step that consumed essentially all of
its budget was genuinely bounded by it, and that `B` is yours to keep
or raise.
- **A step you reimplement** (its Annotation changes). The
measurement describes code you are replacing, so derive the number
afresh. Name the dominant work first -- how many moves, passes,
solver calls or candidate evaluations the new code performs, and how
the cost of one of them scales -- then turn that into seconds by
calibrating against a step of this same program whose measured
runtime you already know. Same machine, same instances, so a measured
anchor is far more reliable than an assumed operations-per-second
rate.

Every step's `Reason` opens with its exact step type followed by a
period, and then says where its number came from: that it is
measured, which step it was calibrated against, why the step was
dropped, or that the step takes the remaining allowance instead of a
number.

### Then allocate

Only `B` is yours to allocate. `C` is whatever the complete work
costs, so a plan of complete-algorithm steps has nothing to
reallocate.

- `sum(Estimated running time) <= {time_limit} seconds`, counting only
the steps that state a number.
- Spend the allowance rather than dividing it evenly. Give more to a
budget-controlled step that is still improving the incumbent, less to
one that is stagnant or disproportionately expensive, and zero to one
you are dropping. Reassign what a shrunk estimate frees to a step
that can use it; when no step can, leave the remainder unspent rather
than padding a step with useless work merely to consume time.
```

Plan-aware code-evolution user-message template (continued)

```
- Never enforce the total in code. The returned source must contain
no global time-limit constant, program-wide deadline, or
remaining-time logic, except the computation that gives a `remaining
time` step its own budget. A step may bound its own work with a local
iteration count, solver limit or timer; that bound may limit only
that step's search intensity, and must never skip the step's
mandatory core or any later step.
- `gurobipy` is the only permitted optimization backend.

{timeout_warning}
```