CODEEVOLVE: an open-source evolutionary framework for algorithmic discovery and optimization

Anonymous ACL submission

Abstract

We introduce CODEEVOLVE, an open-source framework that couples large language models with island-based evolutionary search for end-to-end algorithmic discovery. CODEEVOLVE integrates inspiration-based crossover, meta-prompting, and depth-based refinement on top of a CVT-MAP-Elites archive and a weighted LLM ensemble to generate optimized solutions for complex problems. On the AlphaEvolve benchmark suite, CODEEVOLVE matches or surpasses the reported AlphaEvolve results on 5 of 9 problems and, under matched conditions, outperforms the open-source frameworks OpenEvolve and ShinkaEvolve on 6 of 9. With the open-weight `Qwen3-Coder-30B` backbone, it surpasses the reported AlphaEvolve score on both `CirclePackingSquare` instances at roughly an order of magnitude lower cost than a frontier closed-source ensemble, and remains competitive with EoH on heuristic-design tasks without retuning. Ablations show that the interaction between CODEEVOLVE's components, rather than any single operator, drives these results. We release the framework, experimental data, and practical hyperparameter guidelines at our anonymized project repository.[1]

1 Introduction

The intersection of large language models (LLMs) and algorithmic reasoning has catalyzed a shift from routine program synthesis to automated scientific discovery. While early work demonstrated the ability of transformers to synthesize code from natural language (Chen et al., 2021), recent systems like AlphaCode (Li et al., 2022), AlphaTensor (Fawzi et al., 2022), and AlphaProof (Hubert et al., 2025) have pushed the boundaries of reasoning and are able to discover novel solutions for fundamental problems. However, these systems typically rely on massive, closed-source models and significant computational resources. This dependency has sparked interest in orchestrating smaller, open-weight models (Belcak et al., 2025) to democratize access.

To surpass the limitations of standard prompting, recent research has successfully coupled LLMs with evolutionary algorithms, using models as "mutation operators" that iteratively refine a population of candidate programs. Pioneers like FunSearch (Romera-Paredes et al., 2024) and Evolution of Heuristics (EoH) (Liu et al., 2024) proved this synergy could generate new solutions that outperform human baselines. Building on this foundation, Google DeepMind released AlphaEvolve (Novikov et al., 2025), capable of evolving entire codebases and producing state-of-the-art (SOTA) solutions to a diverse range of problems, yet its closed-source nature and reliance on proprietary models hinders reproducibility.

In this work, we introduce CODEEVOLVE, an evolutionary coding agent that operationalizes LLM-driven search within a transparent, fully open framework. It tackles a meta-optimization problem in which candidate programs are iteratively refined through selection, variation, and recombination guided by execution feedback and fitness signals. Concretely, CODEEVOLVE combines (i) an islands-based genetic algorithm for diversity and parallel search, (ii) a weighted LLM ensemble for model selection, and (iii) three modular operators for exploration and exploitation: inspiration-based crossover, meta-prompting, and depth-based targeted refinement. The key contribution does not lie in the individual components, but rather in the orchestration of these mechanisms into a reproducible search system that balances global exploration, local refinement, and executable feedback.

We evaluate CODEEVOLVE on problems from the AlphaEvolve benchmark (Novikov et al., 2025) and the EoH benchmark suite (Liu et al., 2024). Comparisons with AlphaEvolve and ThetaEvolve use their reported results, while comparisons with

[1] https://github.com/arr2026-submission-throwaway/CodeEvolve-ARR

OpenEvolve and ShinkaEvolve are run under matched configurations and budgets. CODEEVOLVE matches or surpasses reported AlphaEvolve results on 5 of 9 benchmarks and outperforms OpenEvolve and ShinkaEvolve in 6 of 9 problems under matched conditions. On the EoH suite, CODEEVOLVE achieves competitive performance against EoH on heuristic design tasks using only default configurations and a single open-weight LLM. Ablations show that the synergy between CODEEVOLVE's components, rather than any single operator, is the key enabling factor for these results.

Our contributions include:

- An open-source framework for algorithmic discovery that integrates islands-based evolutionary search with modular LLM orchestration, designed for transparency and reproducibility.
- A comprehensive empirical evaluation on established algorithm-discovery benchmarks, including reported closed-source baselines and controlled open-source comparisons under matched budgets, demonstrating strong quality–cost trade-offs with open-weight models.
- An extensive ablation and component-level analysis showing that the synergy between CODEEVOLVE's operators is central to its strongest results, and that no single component suffices in isolation.

2 Related Work

Evolutionary Program Synthesis. Automated program optimization has its roots in Genetic Programming (GP), where populations of code are iteratively refined via crossover and mutation (Koza, 1992; Langdon and Poli, 2013). Recently, LLMs have redefined this field by acting as semantically-aware operators, capable of generating high-quality solutions to complex tasks (Li et al., 2022; Hubert et al., 2025). This paradigm, often termed "Evolution through Large Models" (Lehman et al., 2023; Hemberg et al., 2024), replaces traditional stochastic bit-flipping with informed, language-based reasoning.

Automatic heuristic design (AHD). FunSearch (Romera-Paredes et al., 2024) first demonstrated that coupling LLMs with evolutionary search can discover solutions surpassing human baselines. EoH (Liu et al., 2024) generalized this into a framework for evolving scoring functions or update rules that operate inside fixed algorithmic pipelines and has since become a widely adopted open-source AHD baseline, inspiring many subsequent methods (Ye et al., 2024; Yao et al., 2025; Stein and Bäck, 2025; Liu et al., 2025). AHD frameworks evolve only a single component within a fixed algorithm rather than synthesizing standalone programs end-to-end.

End-to-end algorithmic discovery. Google DeepMind's AlphaEvolve (Novikov et al., 2025; Georgiev et al., 2025) bridges this gap by synthesizing full programs in large-scale scenarios—including GPU kernel optimization and warehouse computing—but remains closed-source. In response, open-source projects such as OpenEvolve (Sharma, 2025), ShinkaEvolve (Lange et al., 2025), and ThetaEvolve (Wang et al., 2025) have emerged to democratize end-to-end algorithmic discovery, with domain-specific variants targeting scaling laws (Lin et al., 2025), cloud scheduling (Cheng et al., 2025), and hardware-specific optimizations (Nagaitsev et al., 2025).

Meta-Optimization. A significant challenge in LLM-driven search is the sensitivity of models to minor prompt variations (Anagnostidis and Bulian, 2024; Leviathan et al., 2025). To mitigate this, researchers have explored meta-prompting—using the LLM to manage its own reasoning instructions (Suzgun and Kalai, 2024; Zhang et al., 2025)—and evolutionary search for prompts, in which natural language strategies for certain problems are iteratively improved to increase the likelihood of an LLM to produce high-performing solutions (Chen et al., 2023; Madaan et al., 2023; Guo et al., 2025; Agrawal et al., 2026).

CODEEVOLVE sits at the intersection of these trends, applying evolutionary strategies not just to solution programs, but to the search process itself. Unlike heuristic-design systems such as FunSearch and EoH, CODEEVOLVE targets end-to-end program discovery, where candidate solutions are complete programs evaluated directly on the task rather than components inserted into a fixed solver. By mirroring the optimization of programs with the evolution of prompts, CODEEVOLVE enables models to reflect on and rewrite their own instructions. This design yields more diverse search trajectories while prioritizing reproducibility and transparent evaluation.

3 Preliminaries

This section introduces the notation and abstractions used to describe the evolutionary process over solutions and prompts.

A **solution** is a program generated to solve a problem. When an existing solution S is used in a prompt to generate a new solution S', we say that S is the *parent solution* of S'. This parent-children relationship imposes a natural forest structure on the solution population, which is a collection of rooted, directed trees. For any solution S, we denote the set of its k nearest ancestors as $A_k(S)$.

A **prompt** is a textual input provided to an LLM to generate a solution. We define the prompt used for generating solution S as its *parent prompt*, denoted by $P(S)$. Since LLMs are inherently probabilistic, a single prompt can generate multiple distinct solutions.

The quality of a solution is quantified by an **evaluation function,** $h : \mathcal{S} \mapsto \mathbb{R}^d$, which maps a solution S from the space of all possible solutions $\mathcal{S}$ to a real-valued vector of performance metrics, such as runtime, memory usage, or objective value.

We also define two **fitness functions** to measure the overall quality of prompts and solutions. The **solution fitness**, $f_{\text{sol}} : \mathcal{S} \mapsto \mathbb{R}_{\geq 0}$, maps a solution to a non-negative score and typically corresponds to the primary metric in h that we aim to maximize. The **prompt fitness**, f_{prompt}, is derived from f_{sol} and is defined as the maximum fitness achieved by any solution generated from that prompt:

$$f_{\text{prompt}}(P) = \max_{S:P(S)=P} \{f_{\text{sol}}(S)\}. \quad (1)$$

This rewards prompts that have demonstrated the potential to generate high-quality solutions, making them valuable candidates for future evolution, even if some of their offspring may be suboptimal.

The primary **optimization goal** is to iteratively evolve an initial population of prompts and solutions, in order to maximize the solution fitness f_{sol} over a maximum number of epochs N, while respecting constraints on other metrics from h, such as execution time and memory usage.

4 Methodology

CODEEVOLVE integrates an evolutionary framework with LLMs to optimize programs. The architecture is based on the island genetic algorithm (Whitley and Starkweather, 1990): multiple populations (islands) evolve independently and periodically exchange their best-performing individuals (migration) according to a predefined topology. This design improves concurrent evaluation, maintains diversity, and propagates successful solutions across the search. At each epoch t, every island i maintains a population of prompts $\mathcal{P}_t^i$ and solutions $\mathcal{S}_t^i$. CODEEVOLVE operates through an iterative process that progressively enhances populations of prompts and solutions via three components. Evolutionary Operators (Section 4.2) generate new individuals, balancing exploration and exploitation. An LLM Ensemble (Section 4.1) provides the generative engine for code modification and recombination. Population Management (Section 4.4) handles evaluation, fitness tracking, migration, and archive updates.

4.1 LLM Ensemble for Solution Generation

The engine behind CODEEVOLVE's solution generation is a weighted ensemble of LLMs—denoted `LLMEnsemble`—that modify and combine preexisting solutions. For each generation task, a model is sampled according to ensemble weights. The sampled model is then prompted to generate new solutions via edits using a diff-based `SEARCH/REPLACE` format: the model identifies a code region and proposes a targeted replacement. Users can configure distinct ensembles for exploration and exploitation (e.g., higher-temperature models for exploration; lower-temperature models for exploitation). In the simplest case, the ensemble consists of a single LLM.

4.2 Evolutionary Operators

New solutions are generated in parallel using exploitation or exploration operators, sampled independently across islands. At each step, one operator is chosen according to an exploration rate p_{explr}, which is governed by a scheduler (Section 4.3):

1. **Depth exploitation.** This operator refines high-performing solutions. A parent S is selected from $\mathcal{S}_t^i$ via rank-based selection, with probability inversely proportional to its rank:

$$\Pr(S) := \frac{\text{rk}(S)^{-1}}{\sum_{S' \in \mathcal{S}_t^i} \text{rk}(S')^{-1}}, \quad (2)$$

where $\text{rk}(S)$ is the position of S when sorting $\mathcal{S}_t^i$ by f_{sol} in descending order. The ensemble is prompted with S, its parent prompt $P(S)$, and its k nearest ancestors $A_k(S)$. This truncated ancestral context encourages targeted, incremental improvements as depth increases, rather than wholesale strategy changes.

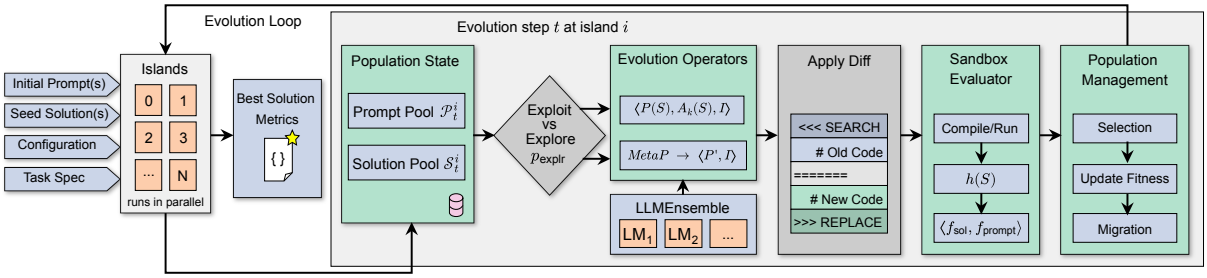

Figure 1: Overview of CODEEVOLVE.

2. **Meta-prompting exploration.** This operator fosters solution diversity and enriches the prompt population with feedback from previous solutions. A solution S and a prompt P are sampled independently and uniformly at random. An auxiliary LLM `MetaPromptingLLM` generates an enriched prompt P' by analyzing P and S. The `LLMEnsemble` uses P' and S to generate a new solution S'. We intentionally exclude the ancestor chain to allow exploration of novel strategies unconstrained by lineage, while leveraging the richer prompt P'.

Inspiration-based Crossover. For both exploitation and exploration, we provide the ensemble with a set of "inspiration" solutions, which are sampled by rank in the former case (Eq. 2) or uniformly at random in the latter case. Because CODEEVOLVE adopts a diff-based `SEARCH/REPLACE` editing format—where the LLM identifies a code region in a single parent program and proposes a targeted replacement—traditional binary crossover is ill-suited: it would require identifying semantically compatible regions across two distinct programs and merging them into a coherent diff, which is highly error-prone. Instead, our inspiration-based approach achieves the semantic intent of crossover more naturally: the LLM is given multiple inspiration solutions alongside the parent, understands the high-level strategies they embody, and incorporates those ideas into the parent via targeted edits. This makes the operator effectively multi-parent, with binary crossover as the special case of a single inspiration.

Algorithm 1 presents the core operator loop. The `LLMEnsemble` receives a prompt, an ancestor set (possibly empty), the target solution, and inspirations, and outputs a new solution. The `MetaPromptingLLM` receives a prompt and a solution and outputs a new prompt. For readability, the pseudocode omits the scheduler and population management logic, as both are orthogonal to the operator logic and described below.

Algorithm 1 Core exploitation/exploration loop of CODEEVOLVE at epoch t

Input: Populations $\mathcal{P}_t^i, \mathcal{S}_t^i$, exploration probability p_{explr}, maximum ancestor depth k
Output: New solution S', and new prompt P' if exploration is chosen
Sample $p \sim \text{Uniform}(0, 1)$
if $p < 1 - p_{\text{explr}}$ **then**
  Sample $S \in \mathcal{S}_t^i$ and inspirations $I \subseteq \mathcal{S}_t^i \setminus \{S\}$ according to Eq. 2
  Collect ancestor solutions $A_k(S)$ from S to its root in $\mathcal{S}_t^i$
  $S' \leftarrow \texttt{LLMEnsemble}(P(S), A_k(S), S, I)$
  $P' \leftarrow \texttt{NULL}$
else
  Sample $S \in \mathcal{S}_t^i$, inspirations $I \subseteq \mathcal{S}_t^i \setminus \{S\}$, and $P \in \mathcal{P}_t^i$ uniformly at random
  $P' \leftarrow \texttt{MetaPromptingLLM}(P, S)$
  $S' \leftarrow \texttt{LLMEnsemble}(P', \emptyset, S, I)$
end if
return S', P'

4.3 Exploration Scheduling

CODEEVOLVE controls the exploration–exploitation trade-off via a scheduler that adjusts the exploration probability p_{explr} over time. In our experiments, we primarily use a *Plateau Scheduler*, which monitors improvements in best-so-far fitness over a sliding window. When progress stalls, the scheduler temporarily increases p_{explr} to encourage exploration; once improvement resumes, it gradually anneals back to a baseline value. We also implement standard decay-based schedules (e.g., exponential or cosine). The scheduler is lightweight, requiring only recent fitness statistics, and operates independently of the evolutionary operators themselves.

4.4 Population Management

CODEEVOLVE includes three mechanisms to manage populations over time.

Initialization. The algorithm begins with an initial solution (often trivial, e.g., a function returning zero) and a basic prompt describing the problem. To create a diverse starting population at each island, the `LLMEnsemble` is prompted multiple times with this initial pair, generating independent approaches that become roots of new solution trees.

Evaluation and Population Control. Each new solution is executed in a sandbox with runtime and memory limits. If execution succeeds, we compute $f_{\text{sol}}(S)$ and metrics $h(S)$ and add S to the population. Failures receive fitness zero, with logs stored for instructive context in future prompts.

Elitist Migration. Top performers from each island are copied to neighboring islands at a fixed migration frequency and rate. To prevent cycles and premature convergence, a solution migrates at most once from its origin island, and we never migrate the best-performing solution of an island in order to preserve its uniqueness. Migrants become roots of new trees upon arrival, with parent pointers set to `NULL`.

Quality–diversity via MAP-Elites. In addition to island populations, CODEEVOLVE maintains a per-island MAP-Elites archive (Mouret and Clune, 2015). Users define feature descriptors (e.g., code-level properties, algorithmic behaviors, or runtime profiles). The archive partitions the feature space either as a regular lattice (MAP-Elites) or via centroidal Voronoi tessellations (CVT-MAP-Elites) (Vassiliades et al., 2017), storing the elite (i.e., most fit) solution per cell. Archive updates occur after evaluation: each successful solution S is mapped to its feature cell, and if S improves the cell's fitness, it replaces the incumbent elite. Sampling for inspirations or parents then draws from the archive to inject structured diversity, e.g., proportional to the rank of all elites in the grid or uniformly over filled cells. The archive thus complements island dynamics by probing diverse niches and reducing premature convergence.

Together, these components are mutually reinforcing: depth exploitation narrows the search along proven lineages, providing stable scaffolding for incremental refinement; inspiration-based crossover prevents premature convergence by importing strategies from distant archive cells that depth exploitation would otherwise never reach; and CVT-MAP-Elites ensures the diversity pool remains populated throughout, so that crossover always has meaningful genetic material to draw from. Our ablations (Section 5) confirm that no single component suffices in isolation—it is their interaction that yields the strongest performance.

5 Experiments

In this section, we are interested in answering the following questions:

RQ1 Can CODEEVOLVE advance the state-of-the-art in automated algorithmic discovery?

RQ2 Can smaller open-weight models compete with more expensive, closed-source LLMs as the backbone of CODEEVOLVE?

RQ3 How do the individual components of CODEEVOLVE contribute to performance and sample efficiency on the proposed benchmarks?

Our primary evaluation is built around a selection of problems from the AlphaEvolve benchmark (Novikov et al., 2025), which defines challenging baselines for automated algorithmic discovery. Apart from comparing CODEEVOLVE against AlphaEvolve itself, which represents the state-of-the-art based on proprietary frontier LLMs, we also compare it against ThetaEvolve (Wang et al., 2025), an RL-based framework that uses open-weight models and is able to produce SOTA solutions to some of the problems at hand. Because AlphaEvolve is closed-source and ThetaEvolve requires large-scale RL infrastructure, these comparisons rely on reported numbers. We complement this with controlled comparisons against OpenEvolve (Sharma, 2025) and ShinkaEvolve (Lange et al., 2025)—the two most prominent open-source frameworks for general algorithmic discovery—under matched configurations and evaluation budgets, assessing CODEEVOLVE's performance in a fair head-to-head setting. We additionally evaluate on the EoH benchmark suite (Liu et al., 2024) to demonstrate breadth on heuristic design tasks.

5.1 Benchmark Problems

We evaluate CODEEVOLVE on problems drawn from two suites. From the AlphaEvolve benchmark (Novikov et al., 2025) we use three packing problems (`CirclePackingSquare`, `CirclePackingRect`, `HexagonPacking`), a point-placement problem (`MinimizeMaxMinDist`), and two autocorrelation-inequality problems

(`FirstAutocorrIneq`, `SecondAutocorrIneq`), yielding nine instances in total. From the Evolution of Heuristics (EoH) benchmark suite (Liu et al., 2024) we use the Online Bin Packing Problem (OBPP), the Traveling Salesman Problem (TSP), and the Flow Shop Scheduling Problem (FSSP). Formal definitions, parameter settings, and pointers to the original implementations are deferred to Appendix A.

5.2 Experimental setup

All experiments were conducted on *AWS SageMaker* with a fixed per-run computational budget in terms of vCPUs and RAM, with each candidate solution being evaluated in an isolated sandbox with strict runtime and memory constraints (see Appendix C for more details). We evaluate two ensemble configurations: one using Google's `GEMINI-2.5` models (Comanici et al., 2025) to approximate a frontier-model setting, and another using only Qwen's `Qwen3-Coder-30B` (Yang et al., 2025) to explore the performance–cost trade-off with open-weight models. Experiments with other LLM backbones can be found in Appendix B.5, and representative prompts and evolved solutions are shown in Appendix D.

5.3 Main Results

Advancing the state-of-the-art. Table 1 compares the best results of CODEEVOLVE against the reported best results of AlphaEvolve and ThetaEvolve. CODEEVOLVE matches or surpasses AlphaEvolve on 5 of the 9 benchmark instances, and establishes new best-known results on `MinimizeMaxMinDist` and `CirclePackingSquare` ($n = 32$). The main gap is on the autocorrelation inequalities, where both AlphaEvolve and ThetaEvolve achieve substantially better results. Appendix B.1 shows visualizations of representative solutions found by CODEEVOLVE and AlphaEvolve (Figures 3-7).

Controlled comparison with open-source frameworks. Table 2 evaluates CODEEVOLVE against OpenEvolve and ShinkaEvolve under identical configurations, evaluation budgets, and LLM backbone (see Appendix B.4 for full details). CODEEVOLVE achieves the best solution in 6 of 9 problems, and the best mean fitness also in 6 of 9 problems, showing strong performance among open-source frameworks under matched conditions.

Comparison with EoH on heuristic design. EoH evolves a single callable component within a fixed algorithmic pipeline using fixed prompt templates—a design optimized for automatic heuristic design (AHD) rather than end-to-end program discovery. Since EoH's architecture makes a direct configuration match infeasible, we compare against EoH's reported results—obtained with GPT-3.5-turbo—while matching the total number of LLM calls and evaluation budget using `Qwen3-Coder-30B`. We ran CODEEVOLVE with 10 islands: 154 epochs for OBPP (30s timeout), 152 for TSP (60s), and 152 for FSSP (120s), reporting best and mean over three independent runs. Complete configurations are available at our anonymized project repository.

Tables 3, 4, and 5 show the comparison. On OBPP, CODEEVOLVE significantly outperforms EoH on the out-of-distribution C500 instances, while EoH leads on C100. On TSP, CODEEVOLVE matches EoH on TSP20 and is slightly behind on TSP50 and TSP100. On FSSP, CODEEVOLVE's best run matches EoH's overall average (0.23 gap) and beats EoH on 6 of 11 Taillard test sets. Taken together with the reported-best comparison in Table 1 and the controlled comparison in Table 2, these results answer **RQ1** in the affirmative.

Cost efficiency. The two ensemble configurations are complementary across problem types. `Qwen3-Coder-30B` achieves the strongest results on the `CirclePackingSquare` instances and both autocorrelation problems, while `GEMINI-2.5` performs best on `CirclePackingRect`, both `HexagonPacking` instances, and both `MinimizeMaxMinDist` instances. The cost difference is most pronounced on `CirclePackingSquare` ($n = 26$): `Qwen3-Coder-30B` surpassed AlphaEvolve after roughly 900 model calls at approximately \$6, whereas `GEMINI-2.5` required approximately 400 calls at $\sim$\$35 (Figures 15-16). This order-of-magnitude cost advantage, combined with competitive performance overall, indicates that open-weight models under modular orchestration are often a viable and cost-effective alternative to frontier proprietary APIs, answering **RQ2** in the affirmative. A full cost and runtime breakdown is provided in Appendix B.3.

5.4 Ablations

We ran ablation studies across all AlphaEvolve benchmark problems using `Qwen3-Coder-30B`. In every case, the full CODEEVOLVE configuration

Table 1: Comparison between best solutions of CODEEVOLVE and reported best solutions of AlphaEvolve (Novikov et al., 2025) and ThetaEvolve (Wang et al., 2025). Best results are in **bold**.

Problem	**AlphaEvolve**	**ThetaEvolve**	**CODEEVOLVE**	
		`Distill-Qwen3-8B`	`Qwen3-Coder-30B`	`GEMINI-2.5 FLASH/PRO`
`CirclePackingSquare`($n = 26$) (↑)	2.63586	**2.63598**	**2.63598**	2.63597
`CirclePackingSquare`($n = 32$) (↑)	2.93794	—	**2.93956**	2.93950
`CirclePackingRect`($n = 21$) (↑)	**2.36583**	—	2.36339	**2.36583**
`HexagonPacking`($n = 11$) (↓)	**3.93009**	—	4.02786	3.93794
`HexagonPacking`($n = 12$) (↓)	**3.94191**	—	4.01092	4.00001
`MinimizeMaxMinDist`($n = 16, d = 2$) (↓)	12.88927	—	12.88925	**12.88923**
`MinimizeMaxMinDist`($n = 14, d = 3$) (↓)	4.16585	—	4.16765	**4.16579**
`FirstAutocorrIneq` (↓)	1.50316	**1.50313**	1.51343	1.55438
`SecondAutocorrIneq` (↑)	**0.96102**	0.94690	0.88110	0.87067

Table 2: Controlled comparison of CODEEVOLVE, ShinkaEvolve (Lange et al., 2025), and OpenEvolve (Sharma, 2025) using `Qwen3-Coder-30B` across three independent runs with matched configurations and evaluation budgets.

Problem	**ShinkaEvolve**		**OpenEvolve**		**CODEEVOLVE**	
	Mean	**Best**	**Mean**	**Best**	**Mean**	**Best**
`CirclePackingSquare`($n = 26$)(↑)	2.47568	2.61551	2.54962	2.6245	**2.63576**	**2.63598**
`CirclePackingSquare`($n = 32$)(↑)	2.53205	2.69473	2.84897	2.93156	**2.93751**	**2.93957**
`CirclePackingRect`($n = 21$)(↑)	2.21694	2.32701	2.28131	**2.36513**	**2.36212**	2.36465
`HexagonPacking`($n = 11$)(↓)	5.55234	5.00000	4.30696	4.20679	**4.02491**	**4.01377**
`HexagonPacking`($n = 12$)(↓)	5.07507	4.38608	**4.01991**	**4.0001**	4.10841	4.05198
`MinimizeMaxMinDist`($n = 16, d = 2$)(↓)	13.48234	13.11627	12.97046	12.94042	**12.91569**	**12.90274**
`MinimizeMaxMinDist`($n = 14, d = 3$)(↓)	4.37160	4.21087	**4.18983**	4.1714	4.26257	**4.16765**
`FirstAutocorrIneq`(↓)	**1.52173**	**1.51556**	1.58031	1.5751	1.54696	1.51642
`SecondAutocorrIneq`(↑)	0.82010	0.84646	0.84634	0.86086	**0.86533**	**0.90114**

produced the best peak performance, with complete plots provided in Appendix B.2 (Figures 8–14). In all such plots, the y-axis shows fitness on a $-\log(M - y + \epsilon)$ scale for increased visual clarity, where M is the maximum fitness attained across all experiments, y is the best fitness at each evaluation step, and ϵ is a small constant.

In the remainder of this section we focus on the `CirclePackingSquare` problem with $n = 32$, which serves as a standard reference benchmark in recent related work (Sharma, 2025; Lange et al., 2025; Wang et al., 2025) and offers an efficient testbed for isolating the contribution of individual components. We report the mean, best and worst outcomes across runs.

Impact of components. We evaluated CODEEVOLVE on three configurations: (i) "full method", which utilizes all operators described in Section 4.2; (ii) "naive evolution", which uses a standard exploration/exploitation pipeline without the proposed components; and (iii) "no evolution", which repeatedly prompts the LLM with the initial prompt and solution, with no contextual data from other solutions. The full method outperforms both alternatives in mean performance and sample efficiency across all benchmarks (Figure 2). For `CirclePackingSquare` ($n = 32$), it is the only configuration that surpasses AlphaEvolve, while for $n = 26$ the naive baseline requires more than twice as many evaluations to match the full method. This shows that, regarding **RQ3**, CODEEVOLVE's components not only increase overall performance, but are the enabling factor for obtaining state-of-the-art results.

Impact of depth and inspirations. We study two key operator hyperparameters: maximum ancestor depth k and number of inspiration solutions ι. Depth-only configurations ($\iota = 0$) do not exceed AlphaEvolve, while inspiration-only ($k = 0$, $\iota = 2, 3$) do. The full method consistently outperforms both in isolation, with higher sample efficiency, indicating a positive synergy between the two operators (Figure 8). Further ablation studies (Appendix B.2) show that MAP-Elites and the choice of migration topology are also key factors for surpassing AlphaEvolve, reinforcing the positive effect of using CODEEVOLVE's components in unison.

Practical guidelines and cross-backbone robustness. The same compact set of defaults proved robust across all benchmarks and the six LLM backbones we tested (Appendix B.5): ring (`Cycle`)

Table 3: Comparison between CODEEVOLVE and EoH's reported results (Liu et al., 2024) for the Online Bin Packing Problem (OBPP). Lower is better.

Method	1k C100	5k C100	10k C100	1k C500	5k C500	10k C500
EoH	**2.24%**	**0.80%**	**0.61%**	2.13%	0.78%	0.61%
CODEEVOLVE (Best)	4.18%	3.16%	3.22%	**0.00%**	**0.20%**	**0.20%**
CODEEVOLVE (Mean)	4.34%	3.34%	3.38%	**0.17%**	**0.37%**	**0.36%**

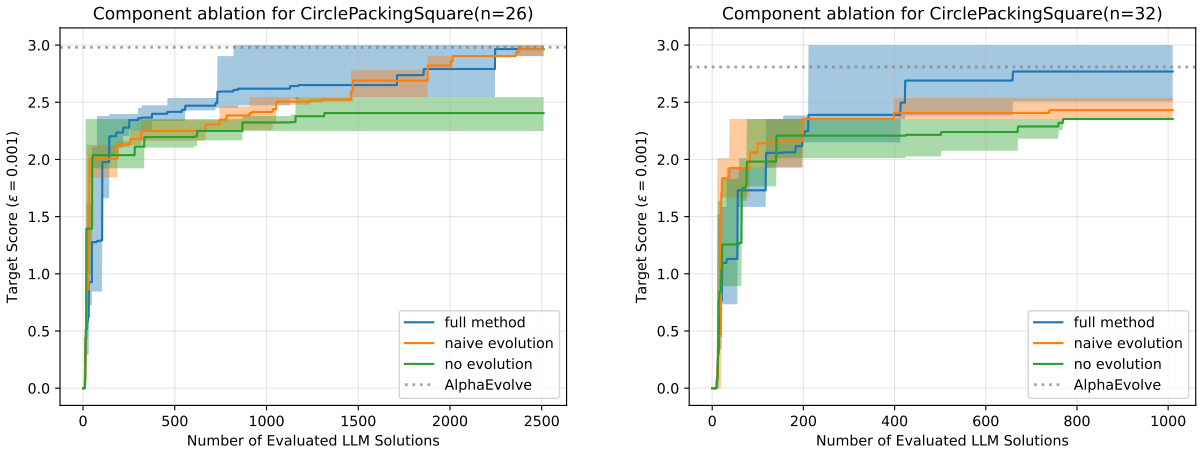

Figure 2: Component ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `CirclePackingSquare` problem. Curves show the mean across three distinct runs, and shaded regions show the best and worst results across all runs.

Table 4: Comparison between CODEEVOLVE and EoH's reported results (Liu et al., 2024) for the Traveling Salesman Problem (TSP). Lower is better.

Method	TSP20	TSP50	TSP100
EoH	**0.000**	**0.000**	**0.025**
CODEEVOLVE (Best)	**0.000**	0.001	0.040
CODEEVOLVE (Mean)	**0.000**	0.011	0.101

Table 5: Comparison between CODEEVOLVE and EoH's reported results (Liu et al., 2024) for the Flow Shop Scheduling Problem (FSSP). Lower is better.

Instance	**EoH**	**CODEEVOLVE**	
		Best	**Mean**
n20m5	0.09	**0.02**	0.33
n20m10	0.30	**0.19**	0.25
n20m20	**0.10**	0.11	0.14
n50m5	**0.02**	0.04	0.05
n50m10	**0.19**	0.27	0.28
n50m20	**0.60**	0.69	0.72
n100m5	-0.04	**-0.05**	-0.03
n100m10	0.14	**0.12**	0.15
n100m20	0.41	**0.40**	0.47
n200m10	**0.12**	0.19	0.23
n200m20	0.61	**0.57**	0.65
Average	**0.23**	**0.23**	0.29

migration topology with migration rate 0.1, initial exploration rate 0.2 with the Plateau Scheduler, CVT-MAP-Elites with fitness and evaluation time as descriptors, maximum ancestor depth $k \in [3, 5]$, and $\iota \in \{2, 3\}$ inspirations. No prompt or operator retuning was needed when switching among `Qwen3-Coder-30B`, `GEMINI-2.5`, `GPT-4.1`, `Claude Sonnet 4.5`, `o4-mini`, and `gpt-oss-20B`, indicating that the orchestration benefits are not specific to a single model family. We provide a full practitioner-oriented discussion of budget-bound parameters, operator-level parameters, and robust defaults in Appendix C.2.

6 Conclusion

We introduced CODEEVOLVE, an open-source framework that democratizes the search for novel algorithms. By integrating an islands-based genetic algorithm with modular LLM operators, CODEEVOLVE bridges the gap between opaque, large-scale systems and accessible research tools. Our experiments demonstrate that CODEEVOLVE matches or surpasses the reported AlphaEvolve results on multiple benchmark instances, compares favorably against open-source evolutionary frameworks under matched conditions, and achieves competitive results against EoH on heuristic design benchmarks despite not being specialized for automatic heuristic design. Perhaps most significantly, our results show that open-weight models, when properly orchestrated, offer a transparent and cost-effective alternative to proprietary APIs.

Limitations

While CODEEVOLVE achieves best-known results on several benchmarks, important limitations remain. First, comparisons with AlphaEvolve and ThetaEvolve rely on reported numbers because AlphaEvolve is closed-source and ThetaEvolve requires RL fine-tuning on infrastructure beyond our budget; differences in evaluation budgets, parallelism, or implementation details may affect these comparisons despite our efforts to match conditions. Second, CODEEVOLVE underperforms on the autocorrelation inequality benchmarks, where both AlphaEvolve and ThetaEvolve achieve substantially better results; this suggests that problems requiring highly specialized analytic constructions may be less amenable to CODEEVOLVE's combinatorial search approach and could benefit from domain-specific adaptations or stronger backbone models. Third, a full ablation grid over problems, components, and LLM backbones is combinatorially expensive; we therefore concentrate the most detailed ablations on `Qwen3-Coder-30B`, while the cross-backbone results reported in Section 5 and Appendix B.5 provide complementary evidence that the architectural benefits are not model-specific. Fourth, the framework introduces hyperparameters (e.g., migration topology, number of islands, number of inspirations, maximum ancestor depth) that may require calibration on novel domains; we provide robust defaults (Appendix C.2), but cannot guarantee they transfer to all problem types. Finally, while CODEEVOLVE significantly reduces costs relative to closed-source alternatives, the inference budget for large-scale evolution remains non-trivial, potentially limiting accessibility for researchers with constrained compute resources.

References

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. 2026. Gepa: Reflective prompt evolution can outperform reinforcement learning. *Preprint*, arXiv:2507.19457.

Sotiris Anagnostidis and Jannis Bulian. 2024. How susceptible are llms to influence in prompts? *Preprint*, arXiv:2408.11865.

Anthropic. 2025. System card: Claude sonnet 4.5. Technical report, Anthropic.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. 2025. Small language models are the future of agentic ai. *Preprint*, arXiv:2506.02153.

Angelica Chen, David M. Dohan, and David R. So. 2023. EvoPrompting: Language models for code-level neural architecture search. In *Advances in Neural Information Processing Systems*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, and others Penedones, Henrique and. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, Jeff Chen, Lakshya Agrawal, Aditya Desai, Jiarong Xing, Koushik Sen, Matei Zaharia, and Ion Stoica. 2025. Barbarians at the gate: How ai is upending systems research. *Preprint*, arXiv:2510.06189.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3290 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *Preprint*, arXiv:2507.06261.

Hamilton Emmons and George Vairaktarakis. 2013. *Flow Shop Scheduling: Theoretical Results, Algorithms, and Applications*. Springer US.

Alhussein Fawzi, Matej Balog, Alexander Huang, Thomas Hubert, Bernardino Romera-Paredes, and 1 others. 2022. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. 2025. Mathematical exploration and discovery at scale. *Preprint*, arXiv:2511.02864.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. 2025. Evoprompt: Connecting llms with evolutionary algorithms yields powerful prompt optimizers. *Preprint*, arXiv:2309.08532.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. 2024. Evolving code with a large language model. *Genetic Programming and Evolvable Machines*, 25(2):21.

Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z. Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, Ottavia Bertolli, Tom Zahavy, Amol Mandhane, Jessica Yung, Iuliya Beloshapka, Borja Ibarz,

Vivek Veeriah, Lei Yu, Oliver Nash, and 20 others. 2025. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*.

John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.

William B Langdon and Riccardo Poli. 2013. *Foundations of genetic programming*. Springer Science & Business Media.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. 2025. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*.

Joel Lehman, Jonathan Gordon, Shreyas Jain, Kenz Ndousse, Christine Yeh, and Kenneth O Stanley. 2023. Evolution through large models. In *Handbook of Evolutionary Machine Learning*, pages 331–366. Springer.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2025. Prompt repetition improves non-reasoning llms. *Preprint*, arXiv:2512.14982.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, and 1 others. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Haowei Lin, Haotian Ye, Wenzheng Feng, Quzhe Huang, Yujun Li, Hubert Lim, Zhengrui Li, Xiangyu Wang, Jianzhu Ma, James Zou, and Yitao Liang. 2025. Can language models discover scaling laws? *Preprint*, arXiv:2507.21184.

Fei Liu, Yilu Liu, Qingfu Zhang, Xialiang Tong, and Mingxuan Yuan. 2025. Eoh-s: Evolution of heuristic set using llms for automated heuristic design. *Preprint*, arXiv:2508.03082.

Fei Liu, Xialiang Tong, Mingxuan Yuan, Xi Lin, Fu Luo, Zhenkun Wang, Zhichao Lu, and Qingfu Zhang. 2024. Evolution of heuristics: towards efficient automatic algorithm design using large language model. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org.

S. Lloyd. 1982. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, pages 46534–46594. Curran Associates, Inc.

Rajesh Matai, Surya Singh, and Murari Lal. 2010. *Traveling Salesman Problem: an Overview of Applications, Formulations, and Solution Approaches*. In-Tech.

Mate Matolcsi and Carlos Vinuesa. 2009. Improved bounds on the supremum of autoconvolutions. *Preprint*, arXiv:0907.1379.

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

Kirill Nagaitsev, Luka Grbcic, Samuel Williams, and Costin Iancu. 2025. Optimizing pytorch inference with llm-based multi-agent systems. *Preprint*, arXiv:2511.16964.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. 2025. Alphaevolve: A coding agent for scientific and algorithmic discovery. *Preprint*, arXiv:2506.13131.

OpenAI, :, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, and 108 others. 2025. gpt-oss-120b and gpt-oss-20b model card. *Preprint*, arXiv:2508.10925.

OpenAI. 2025. Openai o3 and o4-mini system card. Technical report, OpenAI.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

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

Steven S. Seiden. 2002. On the online bin packing problem. *Journal of the ACM*, 49(5):640–671.

Asankhaya Sharma. 2025. Openevolve: an open-source evolutionary coding agent.

Niki van Stein and Thomas Bäck. 2025. Llamea: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2):331–345.

Mirac Suzgun and Adam Tauman Kalai. 2024. Meta-prompting: Enhancing language models with task-agnostic scaffolding. *Preprint*, arXiv:2401.12954.

Vassilis Vassiliades, Konstantinos Chatzilygeroudis, and Jean-Baptiste Mouret. 2017. Using centroidal voronoi tessellations to scale up the multi-dimensional archive of phenotypic elites algorithm. *Preprint*, arXiv:1610.05729.

Yiping Wang, Shao-Rong Su, Zhiyuan Zeng, Eva Xu, Liliang Ren, Xinyu Yang, Zeyi Huang, Xuehai He, Luyao Ma, Baolin Peng, Hao Cheng, Pengcheng He, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. 2025. Thetaevolve: Test-time learning on open problems. *Preprint*, arXiv:2511.23473.

Darrell Whitley and Timothy Starkweather. 1990. Genitor ii.: a distributed genetic algorithm. *J. Exp. Theor. Artif. Intell.*, 2(3):189–214.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, and 41 others. 2025. Qwen3 technical report. *Preprint*, arXiv:2505.09388.

Shunyu Yao, Fei Liu, Xi Lin, Zhichao Lu, Zhenkun Wang, and Qingfu Zhang. 2025. Multi-objective evolution of heuristic using large language model. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(25):27144–27152.

Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. 2024. Reevo: Large language models as hyper-heuristics with reflective evolution. In *Advances in Neural Information Processing Systems*, volume 37, pages 43571–43608. Curran Associates, Inc.

Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. 2025. Meta prompting for ai systems. *Preprint*, arXiv:2311.11482.

A Benchmark Problems

In this section, we provide further details about the problems used to evaluate CODEEVOLVE. Section A.1 covers the four problem categories from the AlphaEvolve benchmark suite (Novikov et al., 2025) (nine instances in total), and Section A.2 covers the three problems from the Evolution of Heuristics (EoH) benchmark suite (Liu et al., 2024). The original EoH implementations are available in EoH's official repository,[2] and our adaptation of these problems for CODEEVOLVE is available in our anonymized project repository.

[2] https://github.com/FeiLiu36/EoH

A.1 AlphaEvolve benchmark

Packing Circles and Hexagons. We consider three packing problems: `CirclePackingSquare`, `CirclePackingRect`, and `HexagonPacking`. The first places n disjoint unit circles in a unit square to maximize the sum of radii ($n = 26, 32$). The second places $n = 21$ circles in a rectangle of perimeter 4. The third packs $n = 11, 12$ unit regular hexagons into a larger hexagon while minimizing its side length.

Minimizing ratio of maximum to minimum distance. This benchmark consists of placing n d-dimensional points in order to minimize the ratio between their maximum to minimum distance, referred to as `MinimizeMaxMinDist`, with instances $n = 16, d = 2$ and $n = 14, d = 3$.

First Autocorrelation Inequality. Let C_1 be the largest constant such that

$$\max_{-1/2 \le t \le 1/2} (f * f)(t) \ge C_1 \left(\int_{-1/4}^{1/4} f(x)dx \right)^2,$$

for all nonnegative functions $f : \mathbb{R} \mapsto \mathbb{R}$, where $f * f$ denotes the convolution operation. Upper bounds on C_1 can be obtained by explicitly constructing step-functions (Matolcsi and Vinuesa, 2009), thus we wish to create an algorithm that generates a nonnegative step function that minimizes the ratio between $\max_{-1/2 \le t \le 1/2}(f * f)(t)$ and $(\int_{-1/4}^{1/4} f(x)dx)^2$.

Second Autocorrelation Inequality. Let C_2 be the smallest constant satisfying

$$\|f * f\|_2^2 \le C \|f * f\|_1 \|f * f\|_\infty,$$

for all nonnegative functions $f : \mathbb{R} \mapsto \mathbb{R}$, where $\|\cdot\|_p$ denotes the p-norm of a given function, and $f * f$ denotes the convolution operation. Hölder's inequality immediately yields $C_2 \le 1$, and mathematicians have been attempting to bound C_2 from below by constructing explicit step functions (Matolcsi and Vinuesa, 2009). The task at hand is thus to create an algorithm that generates a nonnegative step function that maximizes the ratio between $\|f * f\|_2^2$ and $\|f * f\|_1 \|f * f\|_\infty$.

A.2 EoH benchmark

Online Bin Packing Problem (OBPP). The OBPP problem (Seiden, 2002) consists of allocating a collection of items of different sizes into the fewest possible bins with a given fixed capacity,

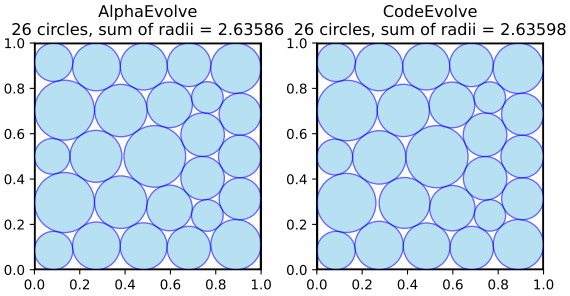

Figure 3: Comparison of best solutions found by CODEEVOLVE and AlphaEvolve for the `CirclePackingSquare` problem with $n = 26$.

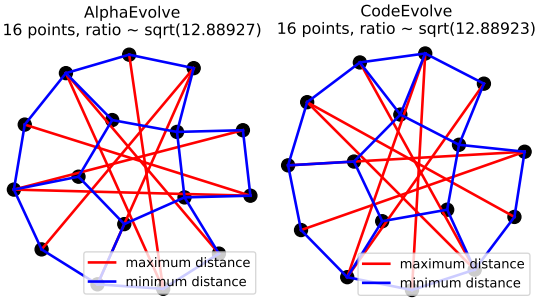

Figure 4: Comparison of best solutions found by CODEEVOLVE and AlphaEvolve for the `MinimizeMaxMinDist` problem with $n = 16, d = 2$.

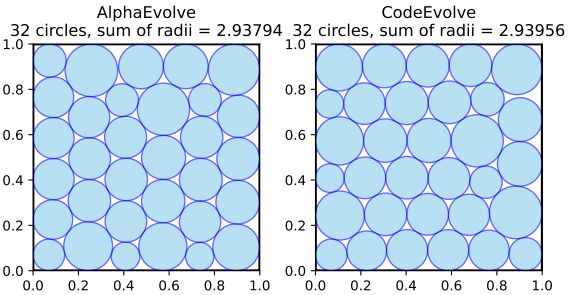

Figure 5: Comparison of best solutions found by CODEEVOLVE and AlphaEvolve for the `CirclePackingSquare` problem with $n = 32$.

where items are packed as they arrive, i.e., we do not have access to all items beforehand. Following (Liu et al., 2024), the fitness is set to be the ratio between a theoretical optimal number of bins and the actual number of bins used, averaged over different instances.

Traveling Salesman Problem (TSP). The TSP problem (Matai et al., 2010) is among the most famous problems in computer science, and consists of finding a Hamiltonian cycle of minimum weight in a given weighted graph. Following (Liu et al., 2024), the aim is to minimize the gap between the solution weight and the optimal solution generated by an exact TSP solver.

Flow Shop Scheduling Problem (FSSP). The FSSP problem (Emmons and Vairaktarakis, 2013) consists of allocating n jobs on m machines, where each job has m tasks that must be performed in a predetermined order on the respective machines. Following (Liu et al., 2024), the objective is to minimize the total schedule length.

B Supplemental experiments and results

B.1 Solution Visualizations

Figures 3–7 display representative best solutions found by CODEEVOLVE and AlphaEvolve for packing and distance optimization benchmarks.

B.2 Further ablations

Component ablations for `CirclePackingSquare`. Figure 2 shows the three-way component ablation (full method, naive evolution, no evolution) on `CirclePackingSquare` for both $n = 26$ and $n = 32$. For $n = 32$, only the full method surpasses the reported AlphaEvolve score, while for $n = 26$ the naive baseline requires more than twice as many evaluations to match the full method. Figure 8 isolates the contributions of depth exploitation and inspiration-based crossover on the $n = 32$ instance: depth-only does not exceed the reported AlphaEvolve score, inspiration-only ($\iota = 2, 3$) does, and combining both yields the highest sample efficiency—confirming the synergy discussed in Section 5.

Component ablations for remaining problems. Figures 9–12 display the component ablations for the remaining benchmark problems not included in Section 5. For the `HexagonPacking`($n = 11$), `FirstAutocorrIneq`, `SecondAutocorrIneq`,

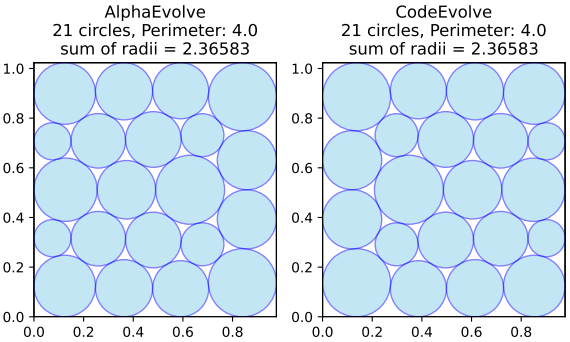

Figure 6: Comparison of best solutions found by CODEEVOLVE and AlphaEvolve for the `CirclePackingRect` problem with $n = 21$.

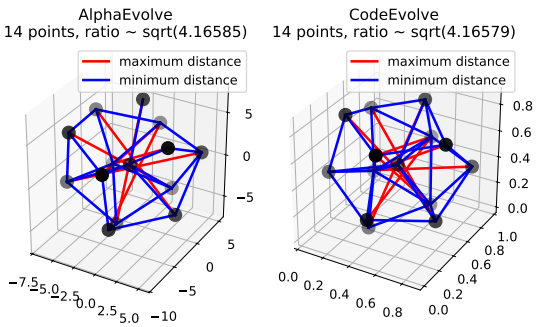

Figure 7: Comparison of best solutions found by CODEEVOLVE and AlphaEvolve for the `MinimizeMaxMinDist` problem with $n = 14, d = 3$.

`MinimizeMaxMinDist`($n = 16, d = 2$) and `CirclePackingRect`($n = 21$) problems, the full method configuration of CODEEVOLVE obtains the superior average and best results across the three runs. For the `HexagonPacking`($n = 12$) and `MinimizeMaxMinDist`($n = 14, d = 3$) problems, although the full method does not present the highest average fitness, it also presents the best solution overall. This further reinforces the positive synergistic effect observed between CODEEVOLVE's components in Section 5, and shows that the full method is the key for producing the best results.

Impact of MAP-Elites. In this experiment, we evaluate the performance of CODEEVOLVE using three distinct elite selection policies: (i) the centroidal Voronoi tessellations MAP-Elites method (Vassiliades et al., 2017), referred to as `CVT`, is a variant of the MAP-Elites (Mouret and Clune, 2015) algorithm, in which the feature space is partitioned according to a fixed number of centroids that approximate a centroidal Voronoi tessellation, e.g., by means of Lloyd's algorithm (Lloyd, 1982); (ii) the traditional MAP-Elites method, referred to as `Grid`, in which the feature space is partitioned according to a regular lattice; and (iii) a naive elite selection, referred to as `Naive`, in which we define a maximum population cap and only add a new solution/prompt if its fitness is greater than the fitness of the worst live individual.

Figure 13 shows the results of this experiment for the `CirclePackingSquare` problem with $n = 32$. The most notable finding is that the MAP-Elites method is clearly necessary for surpassing AlphaEvolve's results, and moreover, the `CVT` variant displays the best performance both in terms of sample efficiency and mean score.

Choice of island topology. In this experiment, we vary the underlying migration topology in order to assess its impact on the performance of CODEEVOLVE. We consider three distinct topologies: (i) the `Cycle` topology connects the N islands according to the undirected cycle graph C_N, i.e., if $\{0, ..., N-1\}$ are the island indices, then island i can send and receive migrants from islands $i-1 \mod N$ and $i+1 \mod N$; (ii) the `Complete` topology connects the islands according to the undirected complete graph K_N, i.e., all islands can send and receive migrants from one another; and (iii) the `Empty` topology does not connect any of the islands, thus suppressing migration.

Figure 14 shows that, for the `CirclePackingSquare` problem with $n = 32$, the `Cycle` configuration is the only one able to surpass AlphaEvolve's results, with the `Complete` topology only being slightly superior to the `Empty` topology in mean score. This shows that migration between islands clearly plays a crucial role in producing state-of-the-art results, but also that excessively migrating between islands can be detrimental, as the overall diversity tends to decrease.

B.3 Cost and Runtime comparison

In this section, we provide further information about the cost and runtime of our experiments. Figures 15 and 16 show the solution and cost histories for both ensemble configurations on the `CirclePackingSquare` problem with $n = 26$ and $n = 32$, respectively. In both cases, `Qwen3-Coder-30B` achieves comparable or superior results to `GEMINI-2.5` at a fraction of the cost.

Table 6 provides approximate costs and runtimes for the best runs of both ensemble configurations on the considered benchmarks. Overall, we can easily see that `Qwen3` is significantly less expensive when compared to `GEMINI-2.5`. The runtime varies between problems, as it mainly depends on the evaluation timeout and number of islands being used (see Table 8), but overall it remains similar between configurations. For the costs and runtimes of all experiments conducted, including the multiple runs done for the ablation studies, see our anonymized project repository.

B.4 Comparison with other open-source frameworks

Table 2 presents the controlled comparison of CODEEVOLVE with OpenEvolve (Sharma, 2025) and ShinkaEvolve (Lange et al., 2025). We adopted

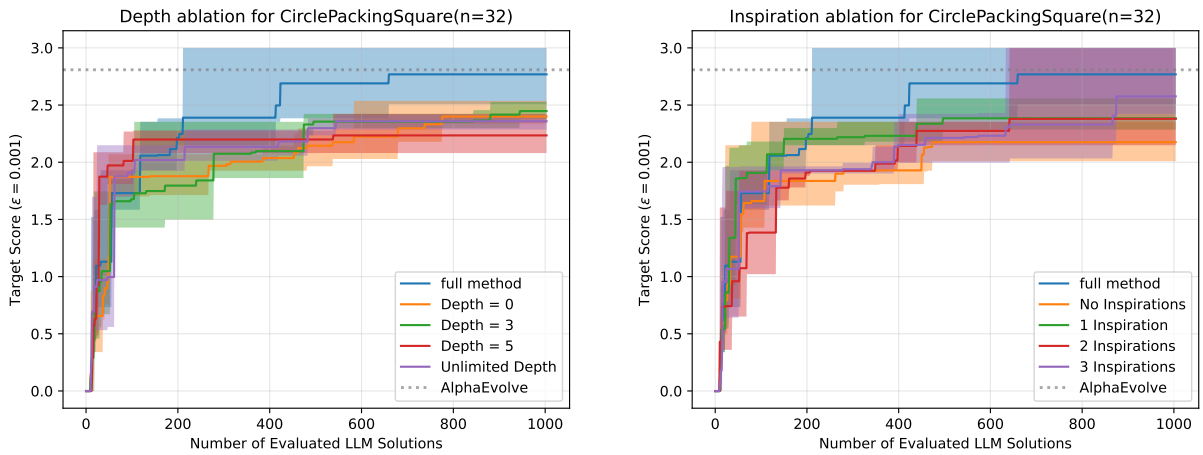

Figure 8: Depth and Inspiration ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `CirclePacking-Square` problem with $n = 32$.

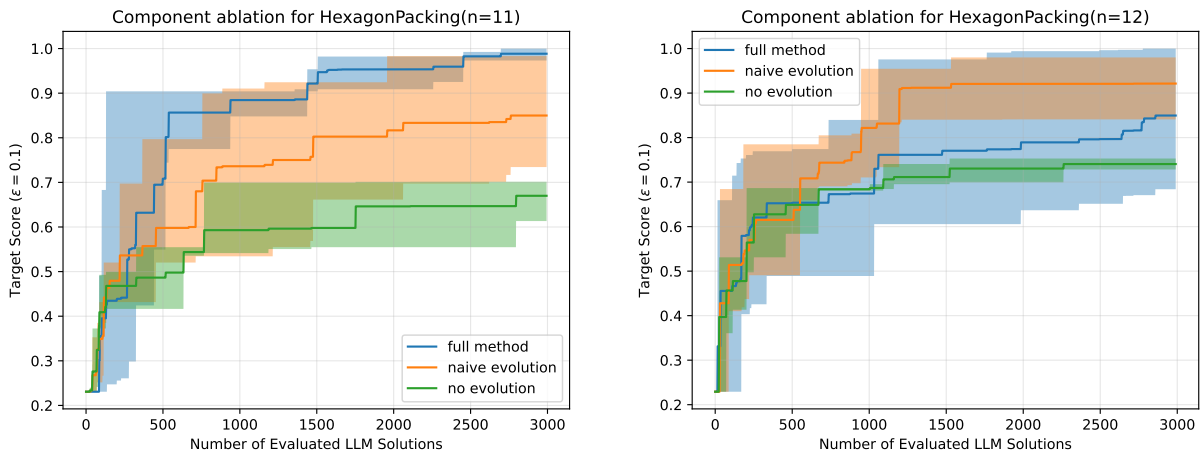

Figure 9: Component ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `HexagonPacking` problem.

the same configurations and evaluation budgets (with minor adaptations to suit each framework) for all benchmark problems. Each framework was run three times with `Qwen3-Coder-30B`, using a default evaluation timeout of 60 seconds and 10 islands. The experimental data and results for both OpenEvolve and ShinkaEvolve are available in our anonymized project repository.

B.5 Experiments with other LLMs

In order to evaluate CODEEVOLVE's sensitivity to the underlying LLM backbone, we test it on four backbones other than the two main ones discussed previously: `GPT-4.1` (OpenAI et al., 2024), `Claude Sonnet 4.5` (Anthropic, 2025), `o4-mini` (OpenAI, 2025) and `gpt-oss-20B` (OpenAI et al., 2025). `GPT-4.1` and `Claude Sonnet 4.5` use the same configuration as the `GEMINI-2.5` ensemble; `o4-mini` and `gpt-oss-20B` use the same configuration as `Qwen3`.

Table 7 reports mean and best performance across three independent runs. `GPT-4.1`, `Claude Sonnet 4.5`, and `O4-MINI` all surpass AlphaEvolve's reported best (2.93794) under the default budget without any retuning. `GPT-OSS-20B` falls short within the default budget, but surpasses AlphaEvolve when given 50 additional epochs (best: 2.93957). These results confirm that CODEEVOLVE generalizes across LLM backbones without requiring retuning of prompts or operator configurations.

C Experiment Details

In this section, we provide further details about the configurations used in our experiments. CODEEVOLVE's components have many different parameters, so we only list the most important ones here.

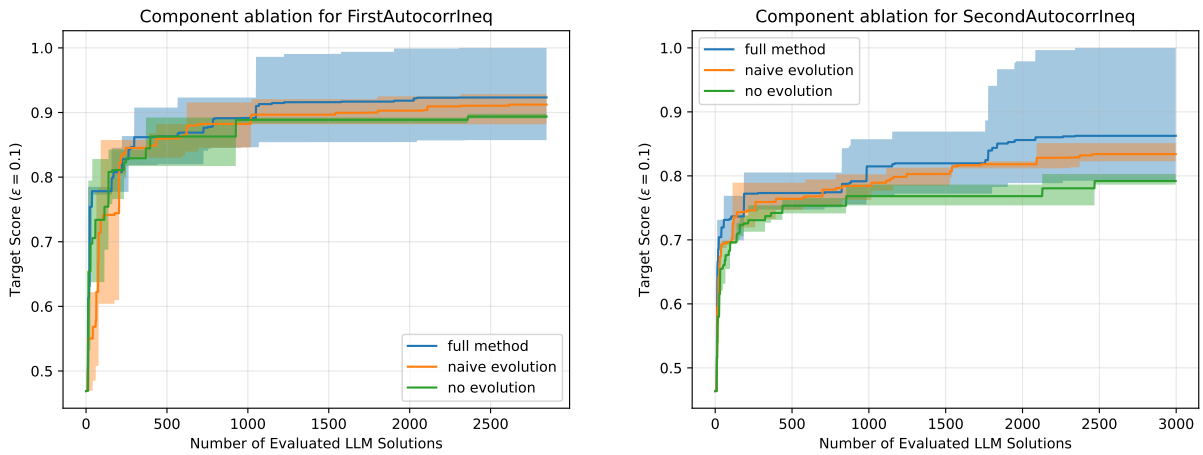

Figure 10: Component ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `AutocorrIneq` problems.

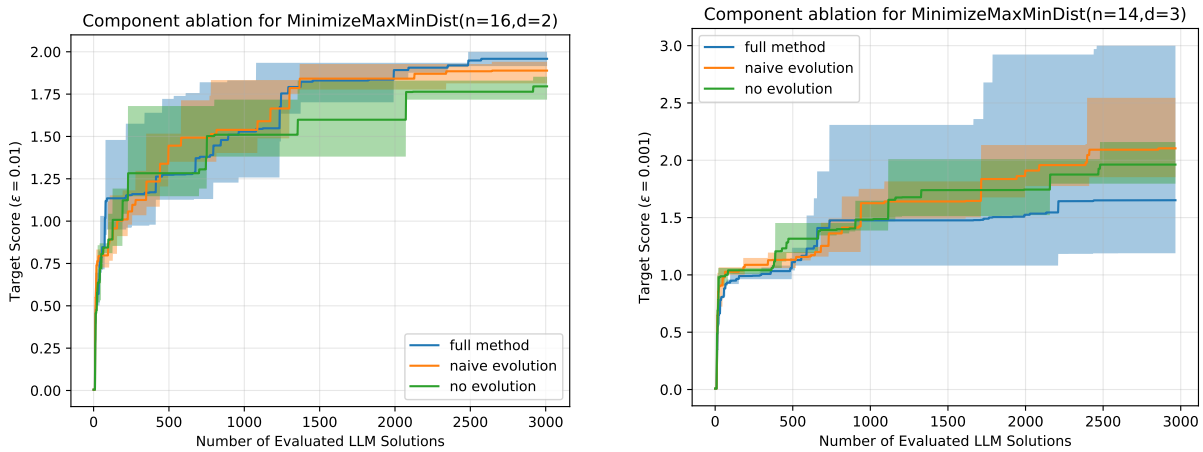

Figure 11: Component ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `MinimizeMaxMinDist` problem.

C.1 Experiment Configurations

All experiments with the `Qwen3-Coder-30B` model use a temperature of 0.7 and top-p of 0.8. The ensemble with `GEMINI-2.5 FLASH/PRO` uses temperatures of 0.7 and top-p of 0.95 for both models. During exploration steps, we only call the `FLASH` variant, and during exploitation steps, we call `FLASH` with 60% probability and `PRO` with 40% probability by default.

Table 8 shows the hyperparameters for the best runs of CODEEVOLVE on the proposed benchmarks (Table 1). By default, we start with an exploration probability of 0.2, and use the Plateau Scheduler to increase this probability by a multiplicative factor of 1.05 if no fitness increase is observed for 5 epochs, and decrease it by 0.95 otherwise, preserving a minimum rate of 0.2 and a maximum rate of 0.5.

C.2 Hyperparameter Sensitivity and Practical Guidelines

CODEEVOLVE introduces several hyperparameters. Below we classify them into three groups and provide practical guidance for new users.

Budget-bound parameters. The number of islands, epochs, evaluation timeout, and memory limit directly trade off solution quality against computational cost. Increasing any of these improves results at higher cost. We found the following defaults to be robust across all tested benchmarks: **5 islands / 200 epochs** for expensive closed-source ensembles (e.g., `GEMINI-2.5`), and **10 islands / 250 epochs** for open-weight models (e.g., `Qwen3-Coder-30B`). Evaluation timeout and memory limits are problem-specific and should be set based on the expected runtime of a single candidate solution evaluation.

Operator parameters. The maximum ancestor depth k and the number of inspirations ι are the

Table 6: Cost and time comparison between `Qwen3-Coder-30B` and `GEMINI-2.5` for the best runs of CODEEVOLVE (Table 1) on the benchmark problems.

Problem	`Qwen3-Coder-30B`		`GEMINI-2.5`	
	Cost (USD)	**Time (Hours)**	**Cost (USD)**	**Time (Hours)**
`CirclePackingSquare`($n = 26$)	6.2	6.4	34.5	6.1
`CirclePackingSquare`($n = 32$)	2.1	2.2	17.2	7.5
`CirclePackingRect`($n = 21$)	10.2	12	67.5	11.1
`HexagonPacking`($n = 11$)	9.1	9.8	73.7	17.5
`HexagonPacking`($n = 12$)	7.4	8.5	77.4	15.6
`MinimizeMaxMinDist`($n = 16, d = 2$)	15.33	25.7	51.5	15.7
`MinimizeMaxMinDist`($n = 14, d = 3$)	9.6	10.9	54.4	13.1
`FirstAutocorrIneq`	23.1	54	70.8	17.3
`SecondAutocorrIneq`	27.2	28.4	66.7	18.7

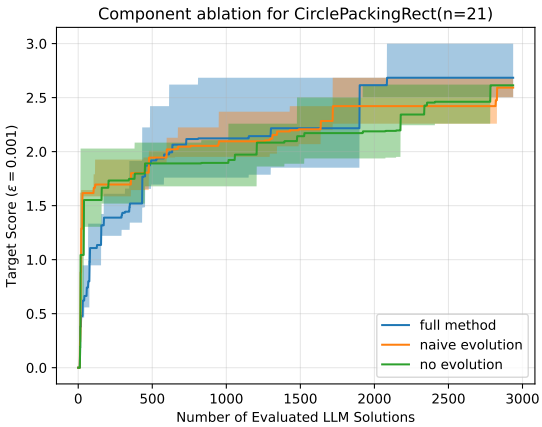

Figure 12: Component ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `CirclePackingRect` problem.

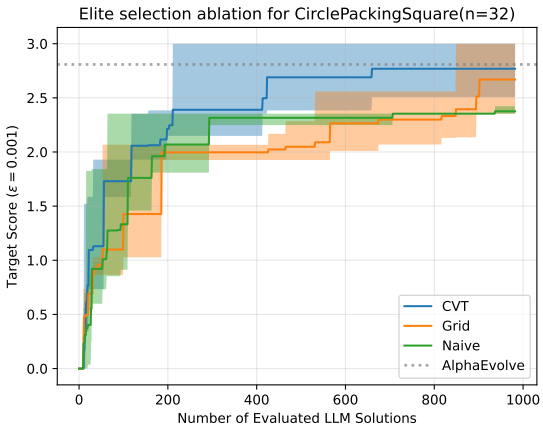

Figure 13: Elite selection ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `CirclePackingSquare` problem with $n = 32$.

Table 7: Performance of CODEEVOLVEwith different LLM backbones for the `CirclePackingSquare` problem with $n = 32$ across three independent runs.

Model	**Mean**	**Best**
`GPT-4.1`	2.93685	**2.93957**
`Claude Sonnet 4.5`	**2.93856**	**2.93957**
`O4-MINI`	2.93767	**2.93957**
`GPT-OSS-20B`	2.93145	2.93535

most sensitive operator-level parameters. Across all tested models and benchmarks, we found $\iota = 2$–3 inspirations and $k = 3$–5 ancestor depth to work well without problem-specific tuning. These values generalized across different backbone models (see Appendix B.5).

Robust defaults. The following settings proved stable across all benchmarks and LLM backbones and serve as safe starting points: migration rate 0.1, initial exploration rate 0.2 with the Plateau Scheduler, ring (`Cycle`) topology, and CVT-MAP-Elites with fitness and evaluation time as feature descriptors. Task-specific meta-prompt customization is optional; a generic template is sufficient to obtain competitive results on new tasks. Complete default configurations are available at our anonymized project repository.

D Examples of prompts and solutions

In this section, we show examples of the initial prompt/solutions and the final best prompt/solution found by CODEEVOLVE for the `CirclePackingSquare` problem with $n = 26$, using `Qwen3-Coder-30B`. For further information about the prompts/solutions, we refer to our anonymized project repository.

Initial prompt:

```
# PROMPT-BLOCK-START

SETTING:
You are an expert computational geometer and
↪  optimization specialist focusing on circle
↪  packing problems.
Your task is to evolve a constructor function
↪  that generates an optimal arrangement of
↪  exactly 26 non-overlapping circles within
↪  a unit square [0,1] × [0,1], maximizing the
↪  sum of their radii.
```

Table 8: Hyperparameters used in the best runs of CODEEVOLVE (Table 1) for the proposed benchmarks.

Problem	Hyperparameters							
	Model	Islands	Epochs	Depth	Inspirations	CPUs	Evaluation (s) Timeout (s)	Evaluation Max Memory (GB)
`CirclePackingSquare`($n = 26$)	`Qwen3-Coder-30B`	10	250	5	2	10	60	1
`CirclePackingSquare`($n = 32$)	`Qwen3-Coder-30B`	10	100	5	2	10	60	1
`CirclePackingRect`($n = 21$)	`GEMINI-2.5`	5	200	5	2	10	60	1
`HexagonPacking`($n = 11$)	`GEMINI-2.5`	5	200	Unlimited	3	10	180	5
`HexagonPacking`($n = 12$)	`GEMINI-2.5`	5	200	Unlimited	3	10	180	5
`MinimizeMaxMinDist`($n = 16, d = 2$)	`GEMINI-2.5`	5	200	Unlimited	3	10	180	5
`MinimizeMaxMinDist`($n = 14, d = 3$)	`GEMINI-2.5`	10	100	Unlimited	3	20	360	5
`FirstAutocorrIneq`	`Qwen3-Coder-30B`	10	500	5	3	10	360	5
`SecondAutocorrIneq`	`Qwen3-Coder-30B`	10	900	5	3	10	60	1

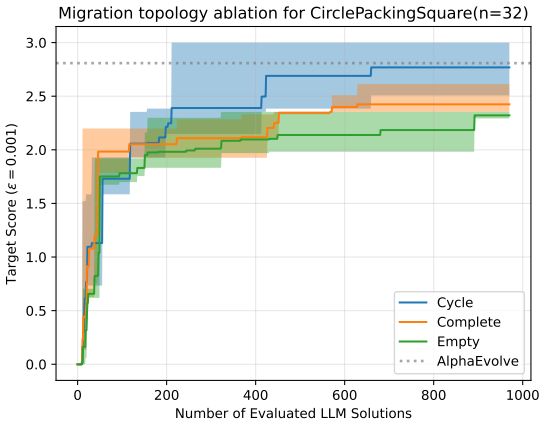

Figure 14: Migration topology ablations for CODEEVOLVE using `Qwen3-Coder-30B` on the `CirclePackingSquare` problem with $n = 32$.

```
PROBLEM CONTEXT:
- Target: Beat the AlphaEvolve benchmark of
↪  sum_radii = 2.6358627564136983
- Constraint: All circles must be fully
↪  contained within the unit square with no
↪  overlaps
- Mathematical formulation: For circle i at
↪  position (xi, yi) with radius ri:
  * Containment: ri ≤ xi ≤ 1-ri and ri ≤ yi ≤
  ↪  1-ri
  * Non-overlap: √[(xi-xj)² + (yi-yj)²] ≥ ri +
  ↪  rj for all i≠j
  * Objective: maximize ∑ri subject to above
  ↪  constraints

COMPUTATIONAL BUDGET:
- **Time limit**: 60 seconds maximum execution
↪  time
- **Memory limit**: 1 GB

COMPUTATIONAL RESOURCES & IMPLEMENTATION
↪  GUIDELINES:
**Core packages**: numpy, scipy, sympy, pandas,
↪  networkx, jax, torch, numba, scikit-learn

**Additional useful packages**:
- **Optimization**: `deap` (evolutionary
↪  algorithms), `platypus` (multi-objective
↪  optimization)
- **Geometric computing**: `shapely` (geometric
↪  operations), `rtree` (spatial indexing),
↪  `scipy.spatial` (KDTree, Voronoi)
- **Constraint programming**:
↪  `python-constraint`, `ortools` (Google
↪  OR-Tools)
- **Physics simulation**: `pymunk` (2D
↪  physics), `pybullet` (physics engine)
- **Performance**: `cython`, `joblib`
↪  (parallelization)

PERFORMANCE METRICS:
1. **sum_radii**: Total sum of all 26 circle
↪  radii (PRIMARY OBJECTIVE - maximize)
2. **benchmark_ratio**: sum_radii /
↪  2.6358627564136983 (progress toward
↪  beating benchmark)
3. **eval_time**: Execution time in seconds
↪  (keep reasonable, prefer accuracy over
↪  speed)

TECHNICAL REQUIREMENTS:
- **Determinism**: Use fixed random seeds if
↪  employing stochastic methods for
↪  reproducibility
- **Error handling**: Graceful handling of
↪  optimization failures or infeasible
↪  configurations
- **Memory efficiency**: Avoid excessive memory
↪  allocation for distance matrix computations
- **Scalability**: Design with potential
↪  extension to different circle counts in
↪  mind

# PROMPT-BLOCK-END
```

Best prompt:

```
# PROMPT-BLOCK-START

SETTING:
You are an expert computational geometer and
↪  optimization specialist focusing on circle
↪  packing problems.
Your task is to evolve a constructor function
↪  that generates an optimal arrangement of
↪  exactly 26 non-overlapping circles within a
↪  unit square [0,1] × [0,1], maximizing the sum
↪  of their radii.
Consider multiple algorithmic paradigms with
↪  emphasis on **efficient**,
↪  **fast-converging**, and **scalable**
↪  approaches that can meet the 60-second time
↪  constraint:
- **Fast heuristic methods**: Greedy placement
↪  algorithms, simulated annealing with smart
↪  cooling schedules, or local search with
↪  adaptive neighborhood structures
```

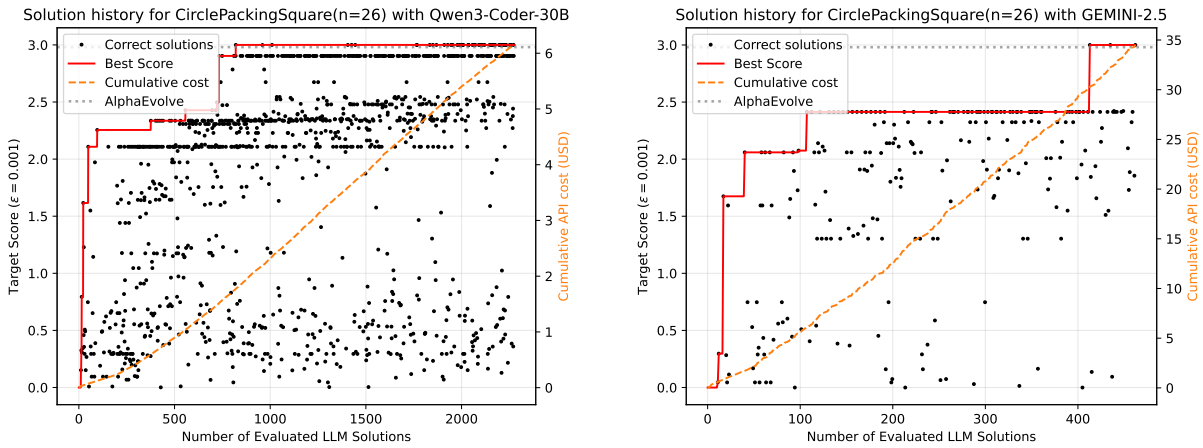

Figure 15: Solution and cost history of `Qwen3-Coder-30B` and `GEMINI-2.5` on `CirclePackingSquare` ($n = 26$). Individual points show the score of solutions executed without errors. The right vertical axis shows cumulative API cost in USD.

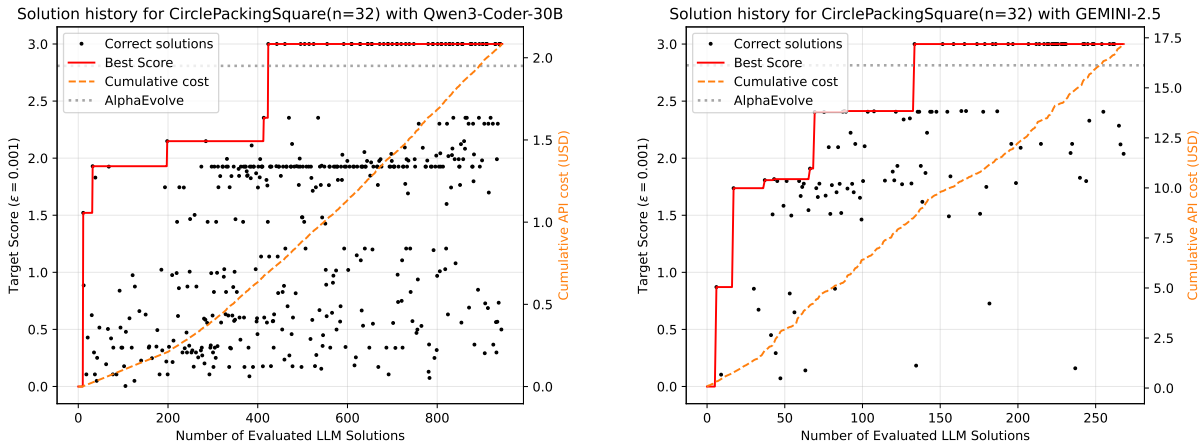

Figure 16: Solution and cost history of `Qwen3-Coder-30B` and `GEMINI-2.5` in the `CirclePackingSquare` problem with $n = 32$.

```
- **Geometric construction heuristics**: Voronoi
↪  diagrams, Delaunay triangulation, or
↪  systematic lattice arrangements with dynamic
↪  refinement
- **Discrete optimization**: Integer programming
↪  formulations, constraint satisfaction with
↪  intelligent pruning, or branch-and-bound
↪  methods
- **Hybrid approaches**: Combine fast geometric
↪  initialization with lightweight optimization
↪  (e.g., coordinate descent, alternating
↪  optimization)
- **Specialized circle packing algorithms**: Use
↪  known patterns like hexagonal close packing,
↪  or adaptations of proven packing strategies
- **Parallel computing**: Implement parallel
↪  evaluation of candidate solutions or
↪  multi-start strategies
- **Approximation algorithms**: Develop provably
↪  good approximations that converge quickly
↪  rather than exact solutions
- **Preprocessing and warm-starting**: Exploit
↪  symmetries, known optimal patterns, or
↪  previous solution information to accelerate
↪  convergence

The key is to prioritize **computational
↪  efficiency** over theoretical optimality,
↪  ensuring the solution can be computed within
↪  the 60-second time limit while still
↪  competing with the benchmark of
↪  2.6358627564136983.

PROBLEM CONTEXT:
- Target: Beat the AlphaEvolve benchmark of
↪  sum_radii = 2.6358627564136983
- Constraint: All circles must be fully contained
↪  within the unit square with no overlaps
- Mathematical formulation: For circle i at
↪  position (xi, yi) with radius ri:
  * Containment: ri ≤ xi ≤ 1-ri and ri ≤ yi ≤
  ↪  1-ri
  * Non-overlap: √[(xi-xj)² + (yi-yj)²] ≥ ri +
  ↪  rj for all i≠j
  * Objective: maximize ∑ri subject to above
  ↪  constraints

COMPUTATIONAL BUDGET:
- **Time limit**: 60 seconds maximum execution
↪  time
```

```
- **Memory limit**: 1 GB

COMPUTATIONAL RESOURCES & IMPLEMENTATION
↪  GUIDELINES:
**Core packages**: numpy, scipy, sympy, pandas,
↪  networkx, jax, torch, numba, scikit-learn

**Additional useful packages**:
- **Optimization**: `deap` (evolutionary
↪  algorithms), `platypus` (multi-objective
↪  optimization), `scipy.optimize` (numerical
↪  optimization)
- **Geometric computing**: `shapely` (geometric
↪  operations), `rtree` (spatial indexing),
↪  `scipy.spatial` (KDTree, Voronoi)
- **Constraint programming**:
↪  `python-constraint`, `ortools` (Google
↪  OR-Tools)
- **Physics simulation**: `pymunk` (2D physics),
↪  `pybullet` (physics engine)
- **Performance**: `cython`, `joblib`
↪  (parallelization)
- **Advanced techniques**: Consider using
↪  `numba.jit` for performance-critical
↪  geometric computations or `jax` for automatic
↪  differentiation in gradient-based methods
- **Specialized solvers**: Consider `cvxpy` for
↪  convex optimization, `gurobipy` for
↪  commercial solvers, or `z3` for SMT solving
- **Machine learning**: Explore `torch` for
↪  neural network approaches or
↪  `stable-baselines3` for reinforcement
↪  learning

PERFORMANCE METRICS:
1. **sum_radii**: Total sum of all 26 circle
↪  radii (PRIMARY OBJECTIVE - maximize)
2. **benchmark_ratio**: sum_radii /
↪  2.6358627564136983 (progress toward beating
↪  benchmark)
3. **eval_time**: Execution time in seconds (keep
↪  reasonable, prefer accuracy over speed)

TECHNICAL REQUIREMENTS:
- **Determinism**: Use fixed random seeds if
↪  employing stochastic methods for
↪  reproducibility
- **Error handling**: Graceful handling of
↪  optimization failures or infeasible
↪  configurations
- **Memory efficiency**: Avoid excessive memory
↪  allocation for distance matrix computations
- **Scalability**: Design with potential
↪  extension to different circle counts in mind
- **Robustness**: Ensure all generated solutions
↪  are mathematically valid (no overlaps, all
↪  circles contained)
- **Alternative approaches**: Prefer methods
↪  that guarantee valid starting configurations
↪  over heuristic initialization methods that
↪  may produce invalid placements

# PROMPT-BLOCK-END
```

Initial solution:

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

# You can define functions outside the main
↪  function below.
# Remember that any function used in parallel
↪  computation must be defined globally and not
↪  locally.

def circle_packing26() -> np.ndarray:
    """
    Places 26 non-overlapping circles in the unit
    ↪  square in order to maximize the sum of
    ↪  radii.

    Returns:
        circles: np.array of shape (26,3), where
        ↪  the i-th row (x,y,r) stores the (x,y)
        ↪  coordinates of the i-th circle of
        ↪  radius r.
    """
    n = 26
    circles = np.zeros((n, 3))

    return circles


# EVOLVE-BLOCK-END
```

Best solution:

```
# EVOLVE-BLOCK-START
import numpy as np
from scipy.optimize import minimize
import math

def circle_packing26() -> np.ndarray:
    """
    Places 26 non-overlapping circles in the unit
    ↪  square in order to maximize the sum of
    ↪  radii.
    Uses a hybrid approach combining geometric
    ↪  initialization and constrained
    ↪  optimization.

    Returns:
        circles: np.array of shape (26,3), where
        ↪  the i-th row (x,y,r) stores the (x,y)
        ↪  coordinates
                of the i-th circle of radius r.
    """
    n = 26

    # Initialize using a better hexagonal layout
    ↪  that starts closer to optimal (from
    ↪  inspiration 2)
    def initialize_better_hexagonal():
        circles = []

        # Use a 5x5 hexagonal grid with optimized
        ↪  spacing
        rows = 5
        cols = 5

        # Calculate spacing more carefully to
        ↪  allow for better optimization
        initial_radius = 0.08  # Slightly smaller
        ↪  initial radius to allow expansion
        spacing_x = 2 * initial_radius
        spacing_y = initial_radius * math.sqrt(3)

        # Center the pattern in the unit square
        offset_x = (1.0 - (cols - 1) * spacing_x)
        ↪  / 2
```

```
    offset_y = (1.0 - (rows - 1) * spacing_y)
    ↪  / 2

    # Generate hexagonal pattern
    for i in range(rows):
        for j in range(cols):
            if len(circles) >= n:
                break
            # Offset every other row
           x_offset = (i % 2) * spacing_x / 2
            x = offset_x + j * spacing_x +
            ↪  x_offset
            y = offset_y + i * spacing_y

            # Add more substantial randomness
            ↪  to avoid getting stuck in
            ↪  local minima
          x += np.random.uniform(-spacing_x/8,
            ↪  spacing_x/8)
          y += np.random.uniform(-spacing_y/8,
            ↪  spacing_y/8)

            # Ensure we're within bounds with
            ↪  safety margin
            x = max(initial_radius, min(1 -
            ↪  initial_radius, x))
            y = max(initial_radius, min(1 -
            ↪  initial_radius, y))

            circles.append([x, y,
            ↪  initial_radius])

    # Trim or pad to exactly 26 circles
    if len(circles) > n:
        circles = circles[:n]
    elif len(circles) < n:
        # Fill remaining slots with strategic
        ↪  positioning
        for i in range(n - len(circles)):
            # Place in center region with
            ↪  more varied positions
           x = 0.2 + 0.6 * np.random.random()
           y = 0.2 + 0.6 * np.random.random()
            r = 0.05 + 0.03 *
            ↪  np.random.random()  #
            ↪  Variable initial radii
            circles.append([x, y, r])

    return np.array(circles)

# Vectorized constraint functions for better
↪  performance (from inspiration 2)
def constraint_containment(circles_flat):
    """Vectorized containment constraints"""
    circles = circles_flat.reshape(-1, 3)
    x = circles[:, 0]
    y = circles[:, 1]
    r = circles[:, 2]

    # All four boundary constraints in one go
    return np.concatenate([
        x - r,            # Left boundary
        1 - x - r,        # Right boundary
        y - r,            # Bottom boundary
        1 - y - r         # Top boundary
    ])

def constraint_nonoverlap(circles_flat):
    """Fully vectorized non-overlap
    ↪  constraints"""
    circles = circles_flat.reshape(-1, 3)

    # Create all pairwise comparisons using
    ↪  broadcasting for maximum efficiency
    n_circles = len(circles)
    if n_circles < 2:
        return np.array([])

    # Extract coordinates and radii
    x = circles[:, 0]
    y = circles[:, 1]
    r = circles[:, 2]

    # Use broadcasting to compute all
    ↪  pairwise distances at once
   # Reshape for broadcasting: (n, 1) and (1,
    ↪  n)
    x_i = x.reshape(-1, 1)
    y_i = y.reshape(-1, 1)
    r_i = r.reshape(-1, 1)

    x_j = x.reshape(1, -1)
    y_j = y.reshape(1, -1)
    r_j = r.reshape(1, -1)

    # Compute squared distances between all
    ↪  pairs
    dist_sq = (x_i - x_j)**2 + (y_i - y_j)**2

    # Compute minimum required distances (sum
    ↪  of radii)
    min_dist_sq = (r_i + r_j)**2

    # We want dist_sq >= min_dist_sq, so we
    ↪  return dist_sq - min_dist_sq
    # But we only want the upper triangle
    ↪  (unique pairs) to avoid duplicates
    mask = np.triu(np.ones((n_circles,
    ↪  n_circles), dtype=bool), k=1)
    constraints = dist_sq[mask] -
    ↪  min_dist_sq[mask]

    # Add a small epsilon to avoid numerical
    ↪  issues with very tight constraints
    # This helps with convergence stability
    return constraints + 1e-12

# Optimization with better convergence
↪  control
def optimize_with_tightening(initial_params,
↪  bounds, constraints):
    """Run optimization with tighter
    ↪  convergence criteria for better
    ↪  results"""
    # Try multiple optimization runs with
    ↪  different tolerances
    methods = ['SLSQP', 'trust-constr']
    best_result = None
    best_sum = -float('inf')

    for method in methods:
        try:
            result = minimize(
                objective,
                initial_params,
                method=method,
                bounds=bounds,
                constraints=constraints,
```

```
                options={'maxiter': 1000,
                ↪  'ftol': 1e-12, 'gtol':
                ↪  1e-12, 'disp': False}
            )

            if result.success:
                current_sum = -result.fun
                if current_sum > best_sum:
                    best_sum = current_sum
                    best_result = result
        except Exception:
            continue

    return best_result

# Multiple restarts with better strategies
↪  (from inspiration 2)
best_result = None
best_sum = -float('inf')

# Try fewer but more strategic restarts (10
↪  instead of 15)
max_restarts = 10

for restart in range(max_restarts):
    # Set seed for reproducibility
    np.random.seed(restart * 1000 + 42)

   # Initialize circles using better approach
    circles = initialize_better_hexagonal()

    # Define optimization bounds: [x1, y1,
    ↪  r1, x2, y2, r2, ...]
    bounds = []
    for i in range(n):
        # x coordinate bounds: [r, 1-r]
        ↪  (ensuring circle fits)
        # y coordinate bounds: [r, 1-r]
        ↪  (ensuring circle fits)
        # r coordinate bounds: [1e-6, 0.4]
        ↪  (tighter upper bound)
        bounds.extend([(1e-6, 0.999), (1e-6,
        ↪  0.999), (1e-6, 0.4)])

   # Flatten initial circles for optimization
    initial_params = circles.flatten()

    # Objective function: minimize negative
    ↪  sum of radii (to maximize sum)
    def objective(params):
        circles_local = params.reshape(-1, 3)
        return -np.sum(circles_local[:, 2])

    # Set up constraints for scipy.optimize
    ↪  using vectorized versions
    constraints = [
        {'type': 'ineq', 'fun':
        ↪  constraint_containment},
        {'type': 'ineq', 'fun':
        ↪  constraint_nonoverlap}
    ]

    # Run optimization with both methods for
    ↪  robustness (from inspiration 2)
    methods = ['SLSQP', 'trust-constr']
    local_best_result = None
    local_best_sum = -float('inf')

    for method in methods:
        try:
            result = minimize(
                objective,
                initial_params,
                method=method,
                bounds=bounds,
                constraints=constraints,
                options={'maxiter': 1000,
                ↪  'ftol': 1e-10, 'gtol':
                ↪  1e-10, 'disp': False}
            )

            if result.success:
                current_sum = -result.fun
                if current_sum >
                ↪  local_best_sum:
                    local_best_sum =
                    ↪  current_sum
                    local_best_result = result
        except Exception:
            continue  # Skip this method if it
            ↪  fails

    # Update global best if we found
    ↪  something better
    if local_best_result is not None and
    ↪  local_best_result.success:
        current_sum = -local_best_result.fun
        if current_sum > best_sum:
            best_sum = current_sum
            best_result = local_best_result

# Return the best result found
if best_result is not None and
↪  best_result.success:
    final_circles = best_result.x.reshape(-1,
    ↪  3)
    # Ensure all circles are properly
    ↪  contained and valid
    for i in range(len(final_circles)):
        x, y, r = final_circles[i]
        # Clamp positions to be within safe
        ↪  bounds
        x = max(r, min(1-r, x))
        y = max(r, min(1-r, y))
        # Clamp radius to reasonable bounds
        r = max(1e-6, min(0.4, r))
        final_circles[i] = [x, y, r]
    return final_circles
else:
    # If all optimizations fail, return the
    ↪  best initial configuration
    return initialize_better_hexagonal()


# EVOLVE-BLOCK-END
```