

CodeEvolve: An open source evolutionary coding agent for algorithm discovery and optimization

Henrique Assumpção^{1,3}, Diego Ferreira^{1,3}, Leandro Campos^{1,3}, Fabricio Murai²

¹Inter Science, Belo Horizonte, MG, Brasil

²Worcester Polytechnic Institute, Worcester, MA, USA

³Federal University of Minas Gerais, Belo Horizonte, MG, Brasil

Correspondence: henrique.soares@inter.co

Abstract

In this work, we introduce CODEEVOLVE, an open-source evolutionary coding agent that unites Large Language Models (LLMs) with genetic algorithms to solve complex computational problems. Our framework adapts powerful evolutionary concepts to the LLM domain, building upon recent methods for generalized scientific discovery. CODEEVOLVE employs an island-based genetic algorithm to maintain population diversity and increase throughput, introduces a novel inspiration-based crossover mechanism that leverages the LLMs context window to combine features from successful solutions, and implements meta-prompting strategies for dynamic exploration of the solution space. We conduct a rigorous evaluation of CODEEVOLVE on a subset of the mathematical benchmarks used to evaluate Google DeepMind’s closed-source AlphaEvolve. Our findings show that our method surpasses AlphaEvolve’s performance on several challenging problems. To foster collaboration and accelerate progress, we release our complete framework as an open-source repository¹.

1 Introduction

The intersection of artificial intelligence (AI) and scientific discovery is entering a new era, marked by a paradigm shift from using AI as a tool for data analysis to using it as automated collaborative agents capable of generating, validating, and experimentally refining novel code solutions. At the forefront of this transformation are Large Language Models, which have demonstrated remarkable capabilities in the generation, understanding, and refinement of complex code (Chen et al., 2021). A framework that leverages the use of LLMs as code-optimization agents can tackle challenging algorithmic problems by exploring the solution space

in a way that closely resembles human creativity and iteration (Romera-Paredes et al., 2023).

Google DeepMind’s AlphaEvolve (Novikov et al., 2025) has recently proven that combining LLMs with evolutionary algorithms can yield state-of-the-art solutions to many complex problems, from improving foundational matrix multiplication algorithms to optimizing large-scale data center infrastructure. AlphaEvolve, however, is described in a closed-source whitepaper that provides few technical implementation details. This opacity presents a significant barrier to progress and hinders collective innovation.

To address these challenges, we introduce CODEEVOLVE, an open-source evolutionary coding agent designed to incorporate the high-level principles of LLM-driven evolution into a concrete and reproducible framework. While inspired by the conceptual vision of AlphaEvolve, CODEEVOLVE proposes new approaches for its core components. Our method integrates an island-based genetic algorithm with a weighted LLM ensemble to iteratively evolve solutions. We design and implement several distinct modular mechanisms to drive this evolution: a depth exploitation operator for targeted refinement, an inspiration-based crossover mechanism to semantically combine features from successful solutions, and a meta-prompting exploration strategy that dynamically boosts the diversity of the prompt population. By incorporating continuous feedback from code execution and performance metrics, this system navigates the complex landscape of program synthesis to discover high-performing solutions.

We can summarize our contributions as follows:

- We propose a novel and transparent framework for automated algorithm discovery that combines both state-of-the-art LLMs and genetic algorithms. We further share both our source code and experimental results as an

¹<https://github.com/inter-co/science-codeevolve>

open-source repository.

- We perform an empirical evaluation of CODEEVOLVE on a suite of complex mathematical problems used to benchmark Google DeepMind’s AlphaEvolve (Novikov et al., 2025), demonstrating that our method outperforms the latter on four distinct problems, yielding new state-of-the-art results.
- In order to understand the impact of the different components of CODEEVOLVE, we present an ablation that validates the contribution of our key architectural components to the overall quality of the solutions.

2 Related Work

Genetic programming and LLMs. Automated generation and optimization of computer programs has long been the domain of Genetic Programming (GP) (Koza, 1992, 1994; Langdon and Poli, 2013), where populations of programs are iteratively improved by operators such as crossover—where multiple programs are combined in order to generate a new one—and mutation—where a program’s content is altered according to some probabilistic process. Although foundational, classical GP methods often struggle with the semantic complexity of modern programming languages. The recent advent of LLMs represents a paradigm shift. With their demonstrated success in generating high-quality solutions for competitive programming tasks (Li et al., 2022), LLMs have emerged as a powerful engine for program synthesis, capable of serving as sophisticated, semantically aware operators for code improvement.

This synergy has given rise to a new class of techniques, formalized as “Evolution through Large Models” (Lehman et al., 2023; Hemberg et al., 2024). The breakthrough application of this concept was FunSearch (Romera-Paredes et al., 2023), which paired an LLM with a programmatic evaluator to discover novel solutions to open problems in mathematics, proving the viability of the approach for genuine scientific discovery. Building on this idea, Google DeepMind recently unveiled AlphaEvolve (Novikov et al., 2025), a closed-source model that generalizes the FunSearch framework from evolving single mathematical functions to entire codebases and a broader range of algorithmic optimization tasks. This system has been successfully in a wide range of different scenarios: from

generating more efficient code for GPU kernels and optimizing the operations of warehouse-scale computers to strengthening results in complexity theory (Nagda et al., 2025). This broad applicability establishes LLM-guided evolution not as a niche technique but as a general-purpose engine for automated problem solving.

Evolutionary coding agents. Following the interest generated by AlphaEvolve, a few open source and independent research projects have focused on developing LLM-driven evolutionary agents. OpenEvolve (Sharma, 2025) provided an accessible implementation of the core features of LLM-driven evolution, confirming its viability for the wider research community. More recently, ShinkaEvolve (Lange et al., 2025) introduced refinements for sample efficiency, employing a bandit-based LLM ensemble and novelty-based rejection filtering to explore the solution space more effectively. CODEEVOLVE falls in this class of LLM-driven evolutionary models, and is designed to be broadly applicable to any algorithmic problem where one wishes to optimize one or more quantifiable metrics, while also aiming to foster collaborative and transparent research.

Meta-prompting. LLMs are notoriously sensitive to prompt variations (Anagnostidis and Bulian, 2024), and a considerable research effort has been directed towards creating systems to automatically design and improve prompts, known in the literature as “meta-prompting” (Suzgun and Kalai, 2024; Zhang et al., 2025). CODEEVOLVE builds upon evolutionary strategies for improving prompts (Fernando et al., 2023; Chen et al., 2023), mirroring the process being executed for the solutions to the underlying problem. By allowing the LLM to reflect on and rewrite its own instructions, it can unlock more diverse and effective evolutionary pathways.

Alternative algorithm discovery paradigms. LLM-driven evolution is part of a broader landscape of AI for scientific discovery, and many distinct approaches have seen major success in recent years. Deep Reinforcement Learning (RL), for instance, has achieved landmark results such as discovering faster matrix multiplication algorithms with AlphaTensor (Fawzi et al., 2022). While incredibly powerful, RL typically requires a more structured environment and a well-defined action space. Other approaches, such as agentic systems, leverage LLMs to reason over scientific hypothe-

ses expressed in natural language (Gottweis et al., 2025). CODEEVOLVE attempts to find a balance between strictness of the former with the more high-level approach of latter, by simultaneously using LLMs to reason about proposed solutions, while also employing a genetic algorithm for a rigorous exploration of the solution space.

3 Preliminaries

The task addressed here constitutes a meta-level optimization: we use an evolutionary algorithm to optimize programs, which themselves solve mathematical optimization problems. To clarify this distinction, this section formally defines the core concepts and notation used throughout the paper.

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 define S as 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 a Large Language Model (LLM) to generate a solution. We denote the prompt used to generate solution S as its parent prompt, $P(S)$. 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 optimize. 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 definition 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 solu-

tions, 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.

4 Methodology

Inspired by Google DeepMind’s AlphaEvolve (Novikov et al., 2025), CODEEVOLVE integrates an evolutionary framework with Large Language Models (LLMs) to optimize programs. The architecture is based on the island genetic algorithm (Whitley and Starkweather, 1990). This model maintains multiple populations (islands), each evolving independently and periodically exchanging their best-performing individuals (migration). The migration step occurs according to a predefined graph (migration topology) whose nodes are islands and whose edges determine the migration paths. This approach enhances the amount of concurrent solutions being evaluated and the overall population diversity, while also allowing for successful solutions to propagate across the entire search effort, thus being a core component of our system. 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 designed to progressively enhance parallel populations of prompts/solutions. This process integrates three key components. The generation of new individuals is driven by a set of **Evolutionary Operators** (Section 4.2), which sample from the population to balance the exploration of novel strategies with the exploitation of promising ones. These operators leverage an **LLM Ensemble for Solution Generation** (Section 4.1) as the underlying engine for all code modifications. The entire evolutionary cycle is orchestrated by the **Population Management** module (Section 4.3), which handles solution evaluation, maintains population fitness, and manages the migration of top-performing individuals between islands.

4.1 LLM Ensemble for Solution Generation

We employ an ensemble of different state-of-the-art LLMs to generate and modify solutions, denoted by LLMEnsemble. Each model in the ensemble is assigned a weight, and for each generation task, a model is sampled according to these weights. Our experiments utilize an ensemble of Google’s GEMINI 2.5 models (Comanici et al., 2025), with

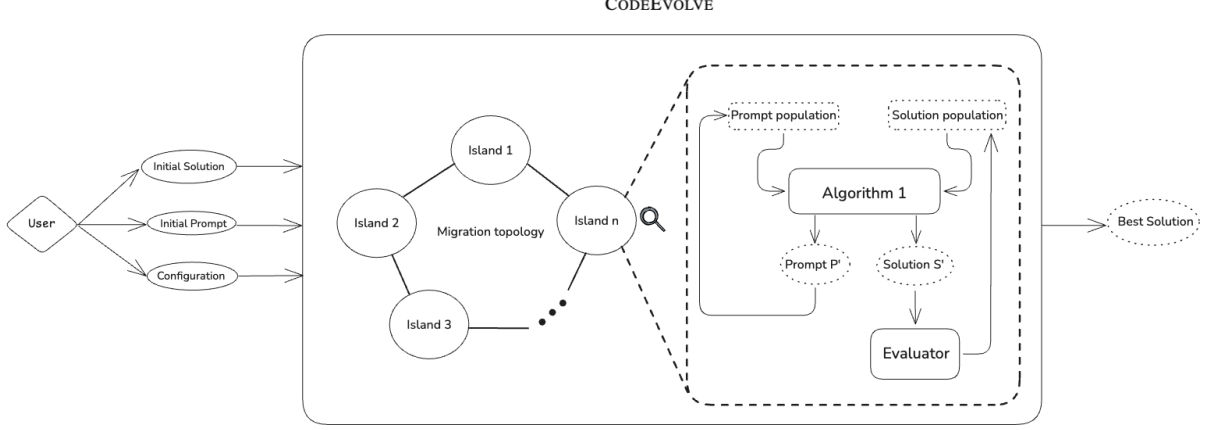


Figure 1: Overview of CODEEVOLVE.

a higher sampling probability for the faster, more cost-effective model (FLASH) and a lower probability for the more powerful, but expensive (PRO). This strategy balances the high-throughput generation of FLASH with the capacity for significant breakthroughs of PRO (Novikov et al., 2025), while also being significantly cheaper, as only using the more expensive model is prohibitive in terms of costs.

4.2 Evolutionary Operators

New solutions are generated using a combination of exploitation and exploration strategies, which are executed in parallel across the islands. At each step, one of these two operators is chosen according to a hyperparameter p_{explr} , which determines the probability of performing the exploration step.

1. **Depth exploitation.** This operator aims to refine high-performing solutions. A parent solution S is selected from the current population \mathcal{S}_t^i via rank-based selection, where the probability of selection is inversely proportional to its rank:

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

where $\text{rk}(S)$ is the position of S when sorting \mathcal{S}_t^i by f_{sol} in descending order. The LLM ensemble is then prompted with S , its parent prompt $P(S)$, and its k closest ancestors, $A_k(S)$. Providing this truncated ancestral context encourages the LLM to perform targeted, incremental improvements rather than generating entirely new approaches as the depth of S in \mathcal{S}_t^i increases.

2. **Meta-prompting exploration.** This operator is designed to foster solution diversity while also enriching the prompt population with feedback from previous solutions. A parent solution S is sampled uniformly from \mathcal{S}_t^i . An auxiliary LLM denoted by MetaPromptingLLM generates a new, enriched prompt P' by analyzing the original parent prompt $P(S)$ and the solution S itself. The LLMEnsemble then uses this new prompt P' and S to generate a new solution S' . Here, we intentionally exclude the ancestor chain to allow the model to explore novel strategies without being constrained by the solution’s direct lineage, while also having access to a prompt P' with richer context.

Inspiration-based Crossover. A key challenge in adapting evolutionary algorithms to LLMs is devising a meaningful crossover operator. Rather than directly splicing code, which often breaks syntax and semantics, we introduce an *inspiration-based crossover* mechanism. For both exploitation and exploration, we sample a set of high-performing "inspiration" solutions from the population using either the rank-based selection policy (if exploiting) or a random sample (if exploring). These complete solutions are provided as additional context to the LLMEnsemble. This encourages the LLM to synthesize new solutions that integrate successful patterns, logic, or functions from multiple parents, effectively performing a semantic crossover within the model’s generative process. In order to prevent premature convergence, we only employ the inspiration-based crossover after the first wave of migrations occurs.

Algorithm 1 shows a pseudocode for the evolu-

tionary operators. The LLMEnsemble (Section 4.1) receives as input a prompt, a set of ancestor solutions (which may be empty), the target solution itself and a set of inspirations, and it outputs a new solution. Similarly, the MetaPromptingLLM described in the exploration step receives as input a prompt and a solution, and outputs a new prompt.

Algorithm 1 Depth exploitation and Meta-prompt exploration steps of CODEEVOLVE at epoch t

- 1: **Input:** Populations $\mathcal{P}_t^i, \mathcal{S}_t^i$, exploration probability p_{explr} , maximum ancestor depth k
 - 2: **Output:** New solution S' , and new prompt P' if exploration is chosen
 - 3: Sample $p \sim \text{Uniform}(0, 1)$
 - 4: Sample a solution $S \in \mathcal{S}_t^i$ (Eq. 2)
 - 5: Sample inspirations $I \subseteq \mathcal{S}_t^i \setminus \{S\}$ (Eq. 2)
 - 6: **if** $p < 1 - p_{\text{explr}}$ **then**
 - 7: Collect ancestor solutions $A_k(S)$ from S to its root in \mathcal{S}_t^i
 - 8: $S' \leftarrow \text{LLMEnsemble}(P(S), A_k(S), S, I)$
 - 9: $P' \leftarrow \text{NULL}$
 - 10: **else**
 - 11: $P' \leftarrow \text{MetaPromptingLLM}(P(S), S)$
 - 12: $S' \leftarrow \text{LLMEnsemble}(P', \emptyset, S, I)$
 - 13: **end if**
 - 14: **return** S', P'
-

In practice, the process of generating new prompts/solutions mentioned above consists of instructing the LLM to modify the parent’s code according to a diff-based SEARCH/REPLACE format, i.e., the LLM first searches for a specific section of code, and then replaces it with the proposed modifications. This ensures targeted and precise edits.

4.3 Population Management

CODEEVOLVE includes three mechanisms to manage the populations of solutions/prompts over time.

Initialization: The algorithm begins with a trivial initial solution (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 a variety of independent approaches that become the roots of new solution trees. This initial diversification has proven crucial for discovering optimal solutions in our experiments.

Evaluation and Population Control: Each new solution is executed in a sandboxed environment

with predefined runtime and memory limits. If a solution runs successfully, its fitness $f_{\text{sol}}(S)$ and other metrics $h(S)$ are computed and it is added to the population. Solutions that fail or exceed resource limits are assigned a fitness of zero, and their execution logs are stored to provide instructive context for future LLM prompts. To maintain a maximum population size, a new individual is added only if its fitness is greater than that of the worst-performing individual currently in the population. The worst individual is then removed from the collection of live individuals.

Elitist Migration: The top-performing solutions from each island are copied and sent to all neighboring islands according to a predefined migration frequency. A migration rate dictates the proportion of solutions to be migrated. To prevent cycles and premature convergence, a solution is only permitted to migrate once from its island of origin. Upon arrival at a new island, a migrant solution is treated as a root of a new evolutionary tree, with its parent pointers set to NULL.

5 Experiments

In this section, we evaluate CODEEVOLVE on a set of mathematical problems from the benchmark suite used to validate AlphaEvolve (Novikov et al., 2025). Specifically, we are interested in answering the following questions:

1. Can CODEEVOLVE advance the state of the art in automated algorithm discovery?
2. How do the different components of CODEEVOLVE impact its performance on the proposed benchmarks?

5.1 Benchmark Problems

We selected four benchmark problems that span different areas of mathematics and algorithm design. Problem P1 comes from real analysis, and problems P2, P3, and P4 are related to geometry.

P1: Second autocorrelation inequality. Let C be the smallest constant satisfying

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

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 $C \leq 1$, and mathematicians have been trying to bound C from below

by explicitly constructing step functions (Matolcsi and Vinuesa, 2009). We are thus interested in obtaining an algorithm to generate nonnegative step functions that maximizes the ratio between $\|f * f\|_2^2$ and $\|f * f\|_1 \|f * f\|_\infty$ (c.f. Novikov et al. (2025, Sec. B.2., Appendix B)).

P2: Minimizing the ratio of maximum to minimum distance. In this problem, we wish to find n d -dimensional points in order to minimize the ratio between their maximum to minimum distance. We consider two instances of this problem: $n = 16, d = 2$, and $n = 14, d = 3$, referred to as P2.A and P2.B, respectively (c.f. Novikov et al. (2025, Sec. B.8., Appendix B)).

P3: Packing circles inside a unit square to maximize sum of radii. In this problem, we wish to place n circles inside a unit square in order to maximize the sum of their radii. We consider two instances: $n = 26$ and $n = 32$, referred to as P3.A and P3.B, respectively (c.f. Novikov et al. (2025, Sec. B.12., Appendix B)).

P4: Packing circles inside a rectangle of perimeter 4 to maximize sum of radii. This problem is a relaxation of P3, where we still wish to place n circles in order to maximize the sum of their radii, but now we allow them to be placed in a rectangle of perimeter 4. We consider the case where $n = 21$ (c.f. Novikov et al. (2025, Sec. B.13., Appendix B)).

5.2 Experimental Setup

All experiments were conducted on the Amazon SageMaker platform for cloud-based computing, with each experimental run utilizing an environment with 8 vCPUs. To ensure fair and reproducible evaluations, we executed each candidate solution in a sandboxed environment with strict resource constraints. For the analysis problem (P1) and the distance minimization problems (P2), solutions were allocated a maximum of 5GB of memory and a runtime limit of 360 seconds. For the circle packing problems, experiments for P3 were allocated 1GB of memory and a 180-second runtime limit, while P4 was given 1GB of memory and a 360-second limit.

Our architecture employed 5 parallel islands for all experiments, each initialized with the same set of starting prompts and trivial solutions. For configurations utilizing the migration mechanism, the 10% top-performing solutions were exchanged be-

| Experiment | AlphaEvolve | CODEEVOLVE |
|------------|----------------|-----------------|
| P1 (↑) | 0.89627 | 0.93768 |
| P2.A (↓) | 12.88926 | 12.88923 |
| P2.B (↓) | 4.16585 | 4.16578 |
| P3.A (↑) | 2.63586 | 2.63596 |
| P3.B (↑) | 2.93794 | 2.93957 |
| P4 (↑) | 2.36583 | 2.36583 |

Table 1: Metrics comparison between CODEEVOLVE and AlphaEvolve on the proposed benchmarks. The symbols \uparrow, \downarrow indicate if we wish to maximize or minimize the problem’s main performance metric, respectively. The best performance on each problem is bold-faced.

tween islands every 40 epochs according to a ring topology, i.e., the underlying graph is the cycle graph C_5 on 5 nodes (islands). The evolutionary process was run for a different number of epochs depending on the problem’s complexity: P1 and P3 ran for 100 epochs, P2 for 200 epochs, and P4 for 150 epochs. All experiments utilized a maximum population size of 40, initial population size of 6, $p_{\text{explr}} = 0.3$. Experiments utilizing the inspiration-based crossover operator selected 3 inspirations from the solution population. All experiments also employed an unlimited maximum ancestor depth k , i.e., we allow all ancestor solutions of a given solution S to be collected in $A_k(S)$ (c.f. Algorithm 1).

As mentioned in Section 4.1, we employ an ensemble of GEMINI 2.5 models. In our experiments, whenever the ensemble is called in CODEEVOLVE, there is an 80% probability of selecting FLASH, and 20% of selecting PRO. Both models are set with temperature parameters equal to 0.7, and top_p to 0.95. The meta-prompting LLM utilizes the same configuration for the FLASH model.

5.3 Results

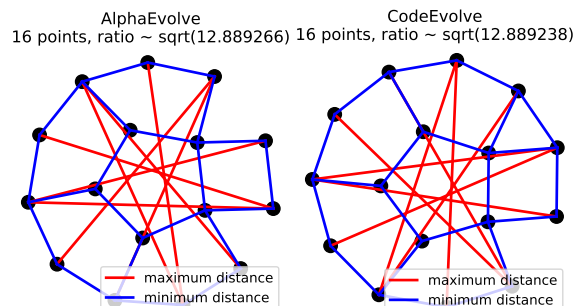


Figure 2: Comparison of best placement of 16 2-dimensional points for problem P2.A.

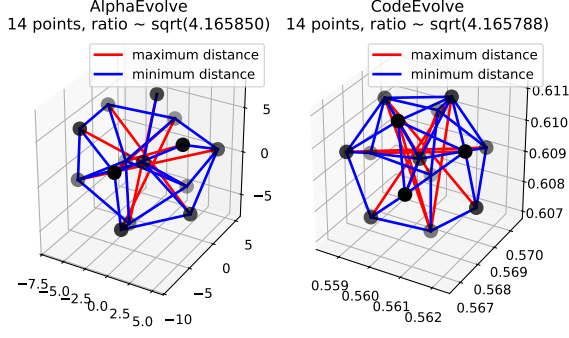


Figure 3: Comparison of best placement of 14 3-dimensional points for problem P2.A.

Table 1 presents a direct comparison between the best results achieved by our CODEEVOLVE framework and the state-of-the-art solutions reported by Google DeepMind’s AlphaEvolve. Our open-source agent not only matches the performance of its closed-source counterpart but surpasses it on five of the six benchmarks. Notably, CODEEVOLVE discovered superior solutions for the second autocorrelation inequality (P1), for both instances of the distance minimization problem (P2.A and P2.B), and for both instances of the circle packing problem within a unit square (P3.A and P3.B). For the final problem of packing circles in a rectangle (P4), CODEEVOLVE achieved an objective value identical to AlphaEvolve’s up to the sixth decimal place, with a minor improvement on the seventh.

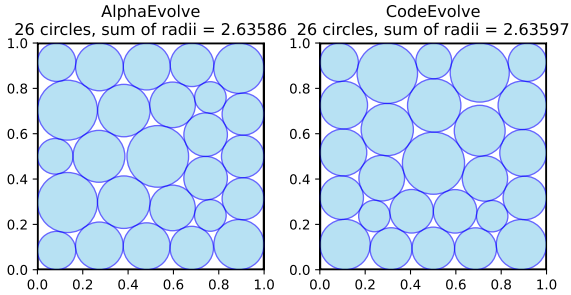


Figure 4: Comparison of best placement of 26 circles inside a unit square for problem P3.A.

To provide a qualitative understanding of our results, we visualize the best-found solutions for the geometric problems. Figures 2 and 3 illustrate the optimal point placements discovered for minimizing the ratio of maximum to minimum distance. Similarly, Figures 4 and 5 display the discovered circle packing configurations within a unit square. The constructions for problems P1 and P4 are displayed in Figures 6 and 7, respectively.

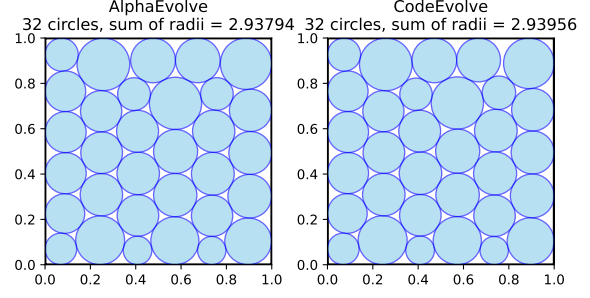


Figure 5: Comparison of best placement of 32 circles inside a unit square for problem P3.B.

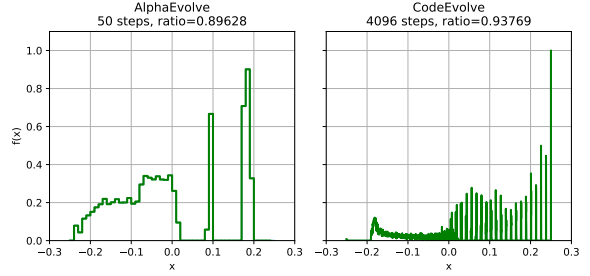


Figure 6: Comparison of best nonnegative step functions found for problem P1. Legend indicates the number of steps in each function, and both functions were normalized so that their image is in $[0, 1]$.

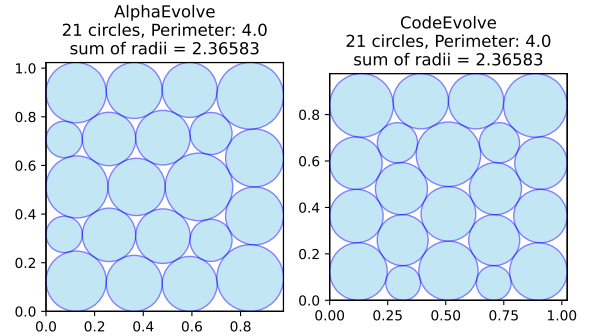


Figure 7: Comparison of best placement of 21 circles inside a rectangle of perimeter 4 for problem P4.

5.4 Ablations

In order to answer the second question posed in this section, we perform an ablation study of the many components of CODEEVOLVE. In particular, we are interested in evaluating the impact of the meta-prompting exploration and inspiration-based crossover operators discussed in Section 4.2. We perform this analysis on problems P1, P2.A and P3 (both instances), according to the different configurations described in Table 2. We choose these three problems in order to cover a diverse set of problems where CODEEVOLVE achieves state of the art performance.

| Configuration | Meta-prompting | Inspirations | Evolution and migration |
|---------------|----------------|--------------|-------------------------|
| mp+insp | ✓ | ✓ | ✓ |
| mp | ✓ | ✗ | ✓ |
| insp | ✗ | ✓ | ✓ |
| no mp or insp | ✗ | ✗ | ✓ |
| no evolution | ✗ | ✗ | ✗ |

Table 2: Configurations for experiments conducted in the ablation study. The no evolution configuration serves as an important baseline, as it involves repeatedly prompting the LLM with only the initial trivial solution, without incorporating any feedback or previously generated code. This isolates the benefit of the iterative evolutionary loop itself.

Figure 8 shows the results of the ablations for problem P3. For problem P3.A, we can observe that all configurations surpass AlphaEvolve’s solution, including the no evolve version without the evolutionary loop. Notably, the full method configuration mp+insp converges to the best-found solution fastest, suggesting a strong synergistic effect between the meta-prompting and inspiration mechanisms. A similar, more pronounced trend is visible in experiment P3.B. Here, the mp+insp configuration again demonstrates the fastest convergence to a superior solution. In contrast, the mp and insp configurations in isolation perform significantly worse, even underperforming the baseline evolutionary model (no mp or insp). This result strongly indicates that for complex packing problems, the combined effect of enriched exploration from meta-prompting and feature mixing from inspiration-based crossover is critical for effective search.

Figure 9 displays the results for problems P1 and P2.A. The results for P2.A align with our previous findings: the full mp+insp configuration outperforms all other instances, followed by insp and mp individually. It is noteworthy that only the configurations including the inspiration mechanism were able to surpass the AlphaEvolve benchmark, highlighting the importance of the crossover-like operator for this geometric problem.

Problem P1, however, presents an interesting exception to the general trend. In this case, the mp configuration achieved the best overall performance, followed by the full mp+insp model. Intriguingly, the addition of inspiration-based crossover appears to have a detrimental effect, as the insp configuration presented the overall worst performance. For this specific task, the broad, diverse exploration driven by meta-prompting may be more effective than the convergence-promoting effect of the inspiration mechanism. This underscores that while

the full CODEEVOLVE architecture is robust, the optimal combination of evolutionary operators can be problem-dependent.

6 Discussion

Progress on other benchmarks. While we have demonstrated state-of-the-art performance on a subset of the AlphaEvolve benchmarks, our work is ongoing. We are actively applying CODEEVOLVE to the remaining mathematical problems outlined in the AlphaEvolve whitepaper (Novikov et al., 2025, Appendix B). For several problems, such as the Heilbronn problem for triangles and packing hexagons, our preliminary results are already approaching the known state-of-the-art bounds. However, other problems, like improving the lower bound for the Kissing Number problem, appear to be significantly harder and may require further methodological innovations. Our goal is to continue pushing the boundaries of what is achievable with a transparent methodology and a limited computational budget. We believe that future algorithmic enhancements to CODEEVOLVE will lead to new breakthroughs.

Reproducibility. We are committed to transparency and reproducibility. Although the sampling operators used in our genetic algorithm are seedable, the main source of stochasticity in CODEEVOLVE lies within the LLM API calls themselves. We utilize an OpenAI-compatible SDK interface, which routes requests to the Google GEMINI models used in our work. As of our experiments, the API does not support fixed random seeds to guarantee deterministic outputs for identical prompts. Consequently, while some components of CODEEVOLVE are reproducible, the exact sequence of generated solutions and the final results cannot be replicated precisely. This is a current limitation of relying on closed-box commercial LLMs

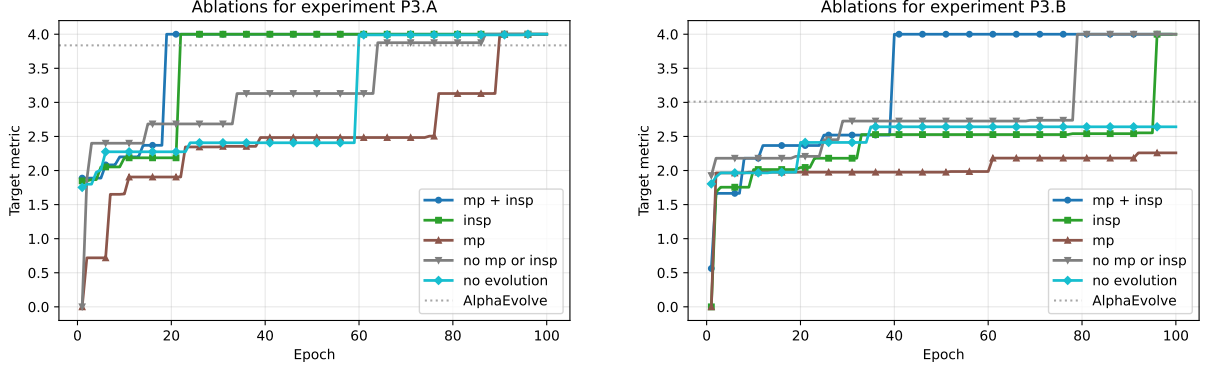


Figure 8: Ablations for both instances of the circle packing problem P3. For easier visualization, the vertical axis shows $-\log(M + \epsilon - y)$, where M is the maximum fitness attained in all experiments, ϵ is a small constant that controls the spacing between curves, and y is the best fitness across all islands at a given epoch.

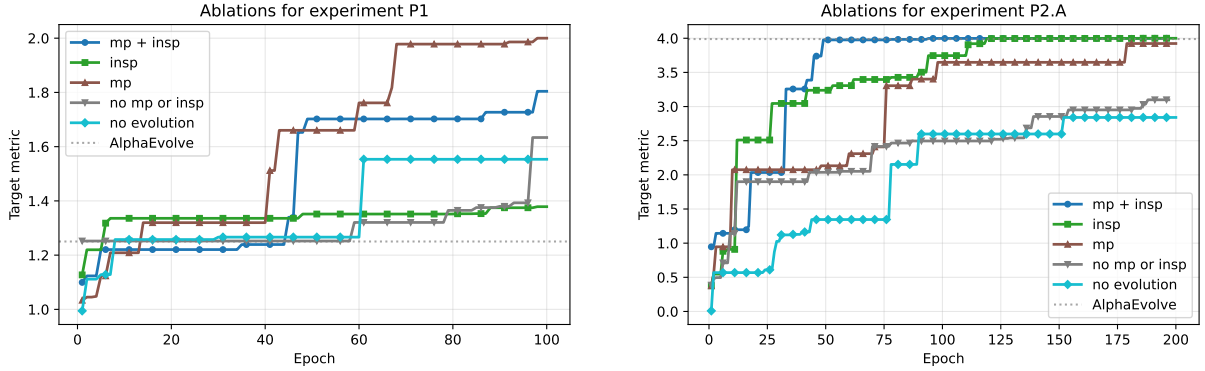


Figure 9: Ablations for problems P1 and P2.A. Vertical axis is defined as in Figure 8.

for scientific research.

Open Source. A significant barrier to progress in LLM-driven algorithm discovery is the closed-source nature of systems like AlphaEvolve (Novikov et al., 2025). This opacity hinders replication, comparative analysis, and collective scientific innovation. Our primary contribution is to address this gap by releasing CODEEVOLVE as a fully open-source project. By providing the complete framework, experimental configurations, and results, we aim to democratize research in this area. We are a small team with limited resources, and we believe that community collaboration is the most effective path toward accelerating progress. We invite researchers and practitioners to build upon our work, introduce new ideas, and collectively tackle challenging algorithmic problems.

7 Conclusion & Future Work

In this paper, we introduced CODEEVOLVE, an open-source framework for algorithm discovery that integrates Large Language Models with an

island-based genetic algorithm. By designing concrete implementations for high-level concepts such as meta-prompting and inspiration-based crossover, we developed a transparent and effective system for evolving codebases. Our empirical evaluation demonstrated that CODEEVOLVE not only replicates but surpasses the performance of the closed-source AlphaEvolve system on several complex mathematical benchmarks, establishing new state-of-the-art results. By releasing our complete framework and experimental results, we provide a valuable resource for the research community, aiming to democratize research in this promising domain.

For future work, we plan to expand upon the current features by incorporating more sophisticated evolutionary mechanisms, such as promoting increased solution diversity via the MAP-Elites algorithm (Mouret and Clune, 2015), and a more dynamic scheduling for the exploration-exploitation trade-off via RL methods. We also intend to conduct a large-scale analysis of hyperparameter sensitivity to further optimize performance across different problem domains. Finally, we will apply

CODEEVOLVE to a broader range of scientific and engineering challenges to continue pushing the boundaries of automated algorithm discovery.

Acknowledgments

The authors thank Bruno Grossi for reviewing this paper and for the continuous support during the development of this project. We also thank Fernando Augusto and Tiago Machado for the useful conversations about possible applications of CODEEVOLVE at Inter.

Author contributions

H.A. started the project, designed and implemented the core features of CODEEVOLVE, and conducted experiments on all of the proposed benchmarks. D.F. implemented the benchmarks related to analysis and the code for the solution evaluator, and also conducted experiments for problem P1. L.C. and F.M. were involved in design discussions about the main components of CODEEVOLVE, as well as the analysis of the experimental results. H.A. and F.M. wrote the manuscript, and D.F. drafted preliminary versions of the two first sections. L.C. and F.M. were responsible for multiple rounds of revisions before the final submission.

References

- Sotiris Anagnostidis and Jannis Bulian. 2024. [How susceptible are llms to influence in prompts?](#) *Preprint*, arXiv:2408.11865.
- 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. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Naveen 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.
- 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.
- Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. 2023. Promptbreeder: Self-referential self-improvement via prompt evolution. *arXiv preprint arXiv:2309.16797*.
- Jonas Gottweis, Wen-Hao Weng, Aleksandr Daryin, Tuan Tu, Anirudh Palepu, and 1 others. 2025. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*.
- 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.
- John R Koza. 1992. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press.
- John R. Koza. 1994. [Genetic programming as a means for programming computers by natural selection](#). *Statistics and Computing*, 4(2):87–112.
- 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.
- 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.
- 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.
- Ansh Nagda, Prabhakar Raghavan, and Abhradeep Thakurta. 2025. [Reinforced generation of combinatorial structures: Applications to complexity theory](#). *Preprint*, arXiv:2509.18057.
- 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.
- Bernardino Romera-Paredes, Mohammad Barekatin, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, and 1 others. 2023. [Mathematical discoveries from program search with large language models](#). *Nature*, 624(7992):545–552.
- Asankhaya Sharma. 2025. [Openevolve: an open-source evolutionary coding agent](#).
- Mirac Suzgun and Adam Tauman Kalai. 2024. [Meta-prompting: Enhancing language models with task-agnostic scaffolding](#). *Preprint*, arXiv:2401.12954.
- Darrell Whitley and Timothy Starkweather. 1990. [Genitor ii.: a distributed genetic algorithm](#). *J. Exp. Theor. Artif. Intell.*, 2(3):189–214.
- Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. 2025. [Meta prompting for ai systems](#). *Preprint*, arXiv:2311.11482.