# Simple Baselines are Competitive with Code Evolution

**Yonatan Gideoni**[*]
University of Oxford
yg@robots.ox.ac.uk

**Sebastian Risi**
Sakana AI and ITU Copenhagen

**Yarin Gal**
University of Oxford

## Abstract

Code evolution is a family of techniques that rely on large language models to search through possible computer programs by evolving existing code. While code evolution pipelines have shown impressive performance across domains, many are highly complex and are typically not compared to simpler alternatives. To prevent bad comparisons from stagnating the field, we propose two simple baselines and compare them to popular pipelines across several domains: finding better mathematical bounds, designing agentic scaffolds, developing heuristic combinatorial optimization algorithms, and machine learning competitions. Surprisingly, the baselines match or outperform existing methods across all domains, prompting a closer look at which factors drive performance in different settings. For mathematical bounds, we find that the search space matters far more than the exact search algorithm, with both simple and complex methods performing similarly given the same search space. Thus, expanding the search space is what matters for improving performance, and does so across all pipelines. Moreover, different domain knowledge embedded in prompts can significantly affect the search's efficiency, potentially leading to overestimating a pipeline's sample efficiency. For automated agentic scaffold design, we show that typical high-variance evaluations lead to all automated search methods picking subpar scaffolds relative to a simple majority vote. To mitigate this, we propose improved evaluation procedures that reduce stochasticity while keeping the search economically feasible. Overall, these results indicate that further improving code evolution's performance depends more on the setup surrounding the search pipeline than on the pipeline itself. We hope these insights will enable developing new code evolution methods and help spur meaningful progress in the field.

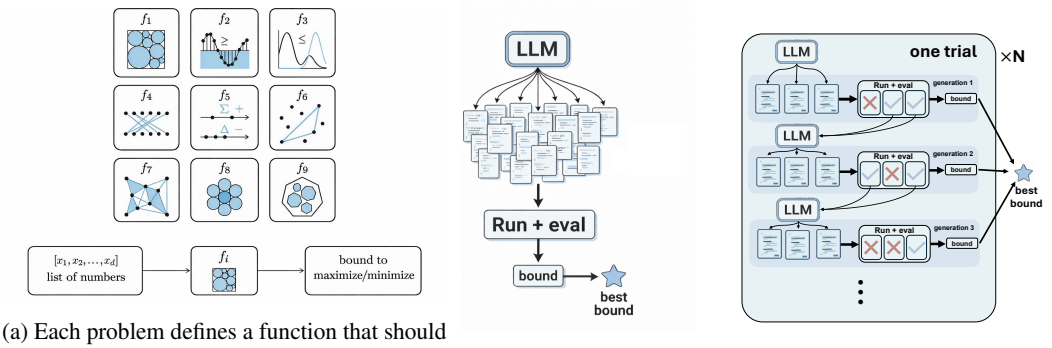
## 1 Introduction

For many problems, a solution can take the form of a computer program, thus allowing automated program search to solve a variety of tasks. This has been demonstrated across several domains: for mathematical and algorithmic discovery, Novikov et al. [23] find programs that improve on several mathematical bounds. Hu et al. [15] use LLMs to design programs defining agentic scaffolds which solve math and science problems. Chan et al. [6] test how well LLM-based automated coding methods can perform in machine learning competitions.

Many systems search over code space by feeding programs into language models and asking them to evolve, crossover, and recombine them, thereby building LLM pipelines that do code evolution. However, these pipelines consist of numerous design choices, including using ensembles of language models for diversity and cost efficiency, automated parent program selection to maximize diversity, and various ways of receiving feedback [20, 23, 28]. These many design choices are often not ablated, nor are the pipelines compared to simpler alternatives.

[*]Work partially done during an internship at Sakana AI. Code is available at https://github.com/YonatanGideoni/code_evo_simple_baselines.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).




(a) Each problem defines a function that should be maximized/minimized.

(b) IID random sampling

(c) Sequential conditioned sampling

Figure 1: **(a)** For AlphaEvolve-style mathematical bounds each problem effectively defines a function that gets as input a list of numbers, possibly with some extra structure, and outputs a mathematical bound that should be maximized or minimized. **(b, c)** The two baselines, **(b)** randomly sampling a set of programs IID from an LLM and picking the best one and **(c)** generating a set of programs, evaluating them, and generating a new set conditioned on some of the programs that ran successfully.

Historically, bad comparisons have threatened to stagnate various subfields. In interpretability, Adebayo et al. [3] showed that many existing image saliency methods give similar saliency maps to edge detectors and are independent of the model or the data. Studying drivers of LLM reasoning capabilities, Shao et al. [27] find that some perceived improvements were due to using models that benefit even from random rewards. Henderson et al. [14] showed various shortcomings in how deep reinforcement learning (RL) was evaluated, such as high stochasticity and different implementations leading the same method to give meaningfully different results. Agarwal et al. [4] further demonstrate that reliance on summary statistics across RL runs results in non-robust evaluations, rendering seemingly strong results unreliable.

To prevent code evolution from suffering similar shortcomings, we ask: *what matters for code evolution?* To systematically test this, we propose two simple baselines – random sampling from the language model and a more sequential version thereof – and compare them to several existing pipelines over four different domains.[2] Each domain tests how methods perform under a different budget constraint, such as a limited API budget when finding mathematical bounds [23], the number of function evaluations when designing agentic scaffolds [15] or finding heuristics for combinatorial optimization problems [17], or wall-clock time in machine learning competitions [6]. All domains are tested using either the standard or above-standard budgets in the literature.

Surprisingly, in all settings and under the same budget constraints, at least one of the baselines matches or exceeds a purpose-built code evolution pipeline. To understand why, we look into what matters for these search processes. When searching for mathematical bounds, simple and complicated pipelines seem to find high-performing bounds similarly well (§4.2). Thus, changing the search space – and thereby the performance ceiling – improves the discovered bounds, which simple and complex pipelines similarly manage to find. This implies existing code evolution systems perform a confined open-ended search, whereas true open-endedness matters much more for improving performance (§4.3). Moreover, domain knowledge embedded in prompts can substantially alter a search's efficiency, resulting in a pipeline mistakenly seeming more efficient than another (§4.1). When searching for agentic scaffolds, we find that all automated search methods suffer from non-robust high-variance summary statistics, leading subpar scaffolds to be selected (§5). To mitigate this, we propose new robust evaluations that reduce stochasticity while keeping the search economical (§5.1).

Notably, the search space, domain knowledge, and evaluation methodology are typically not evolved but manually designed by method developers or domain experts. Thus, further performance improvements would likely stem from components external to the search pipeline, which in existing systems are typically kept fixed. We conclude by discussing implications and a set of best practices for future work (§8).

[2] Although the baselines are also a form of code evolution, "code evolution" here generally refers to more complicated methods unless specified otherwise.

## 2 Background

We briefly present code evolution here, with additional related work delegated to Appendix A. Code evolution pipelines typically work by starting with some program and using an LLM to generate new, improved versions of it. Previous programs are often given in the LLM's prompt, selected to maximize a mix of diversity and fitness. This is broadly how systems like AlphaEvolve [23], ShinkaEvolve [20], OpenEvolve [28], and ADAS [15] work. Some of these systems have many components, often requiring thousands of lines of code.

One example application of code evolution is in finding bounds for the circle packing problem. There, one must maximize the sum of $n$ circle radii while ensuring all circles fit in a unit box with no mutual overlap. Given a list of numbers defining centers and radii, it is straightforward to automatically check whether they define a valid packing, thereby giving a new bound, with this process illustrated in Figure 1a. Finding a better bound is thus reduced to finding a list of numbers with some properties. Code evolution designs functions that output such lists, either by searching over the numbers and returning the best found candidate or by constructing these lists directly.

Previous works using code evolution for discovery mention how changing a problem's search space or using domain expertise can improve a search's performance and efficiency but do not deeply discuss it. Novikov et al. [23] discuss how occasionally injecting human priors into their code helped them find better matrix multiplication algorithms. Georgiev et al. [10] present a large-scale case study using AlphaEvolve in conjunction with other tools to find better bounds over 67 problems, noting that a good formulation or telling a pipeline about a theorem can make the difference between finding an improved bound or the pipeline getting stuck.

In some code evolution applications, simple pipelines similar to the baselines presented here are widely used. For ARC AGI, many of the best-performing methods are based on selecting the best program from a large set of generated candidates, perhaps with some additional domain-specific tweaks, e.g. Greenblatt [13]. Because we focus on whether the additional complexity in some code evolution systems is beneficial, we do not compare to these simple pipelines.

## 3 Baselines for Code Evolution

The two baselines are illustrated in Figures 1b and 1c. The first baseline, IID random sampling (**IID RS**), consists of prompting a language model to produce code that solves some task. After evaluating all programs the best candidate is picked, based on the best-found bound for mathematical discovery or on a validation performance for the other problems.

The second baseline is a modification of IID RS, designed to better deal with problems requiring iterative improvements. For example, in the kissing problem, one of the problems tackled in Novikov et al. [23], one must find the largest possible set of vectors satisfying some property. LLMs often construct the solution directly by specifying these vectors and, at least in their initial answer, do not extend known constructions. Thus iterative methods would likely perform better on similar problems.

Sequential conditioned sampling (**SCS**) works by first generating a set of programs, akin to IID RS, and then generating another by conditioning on a random subset of the programs that successfully ran in the previous generation. Thus, both baselines have no explicit fitness-based selection. This is repeated for a few generations, and optionally then restarts from scratch. SCS' sequentiality comes at the cost of having each program be more expensive to generate as it results in much longer prompts and is less parallelizable than IID RS. The loss of parallelism is potentially significant in cases where the main bottleneck is wall-clock time and not API cost, function evaluations, or compute.

Operationally, there are a few ways to measure complexity. Typical code evolution systems like ShinkaEvolve have tens of unique hyperparameters, while IID RS has none and SCS has three. This is disregarding shared hyperparameters, like the language model's sampling temperature. Another rough metric for complexity is code length – both baselines take at most a single file of Python to fully write out, while complex code evolution codebases are often much larger. When we refer to the baselines being "simple", operationally we mean simple with respect to these metrics.

When comparing methods, different pipelines can arrive at similar solutions or reach the same apparent performance ceiling. We consider two results to be equal if they are the same as per `numpy.isclose`, with default relative and absolute tolerances, as solutions may differ due to numerical precision or otherwise small numerical instabilities. This is important as for some tasks, like machine learning competitions or when finding mathematical bounds, seemingly small improvements can still be meaningful.

## 4 Finding Mathematical Bounds

To test how the baselines compare to code evolution, we test how well they can find mathematical bounds for nine problems from Novikov et al. [23]. These problems are subdivided into those belonging to analysis, combinatorics, or geometry. We sample three, two, and four problems from each category respectively, with the slight imbalance due to combinatorics having only two problems. Brief summaries of the problems and their bounds are in Table 3 in Appendix B, with longer explanations in Novikov et al. [23].

As AlphaEvolve is closed-source, we compare to a similar open-source sample-efficient pipeline, ShinkaEvolve [20]. For a fair comparison, all methods use Gemini-2.5 models, so differences in performance are not due to different model families. ShinkaEvolve is restricted to Gemini-2.5 Pro, Flash, and Flash Lite, akin to AlphaEvolve, while both baselines use Gemini-2.5 Pro. Note that the Gemini-2.5 family's knowledge cutoff is January 2025, several months before AlphaEvolve came out, so good performance is not due to memorizing known answers [8]. Additional technical details are in Appendix C.

For all methods reported results are for a $20 budget. Under these conditions, ShinkaEvolve runs for 500-800 generations, or 25-40 generations per dollar. For reference, the circle packing run in Lange et al. [20] costs about $12 and ran for 150 generations (12.5 generations per dollar). Specifically for estimating uncertainties beyond-budget baseline runs were used, see Appendix G for details. Only a single ShinkaEvolve run is used per problem as the experiments are very expensive – running ShinkaEvolve and the two baselines altogether costs >$70 per problem. Some of ShinkaEvolve's hyperparameters were slightly tuned to improve its performance, see Appendix C for details.

### 4.1 Different Methods Should Use the Same Prompts: Domain Knowledge Significantly Affects Sample Efficiency

```
You are an expert mathematician specializing in circle packing problems and computational
↪  geometry. The best known result for the sum of radii when packing 26 circles in a unit
↪  square is 2.635.

Key directions to explore:
1. The optimal arrangement likely involves variable-sized circles
2. A pure hexagonal arrangement may not be optimal due to edge effects
3. The densest known circle packings often use a hybrid approach
4. The optimization routine is critically important - simple physics-based models with
↪  carefully tuned parameters
5. Consider strategic placement of circles at square corners and edges
6. Adjusting the pattern to place larger circles at the center and smaller at the edges
7. The math literature suggests special arrangements for specific values of n
8. You can use the scipy optimize package (e.g. LP or SLSQP) to optimize the radii given
↪  center locations and constraints

Be creative and try to find a new solution better than the best known result.
```

Listing 1: The default circle packing prompt used by ShinkaEvolve. In addition to broadly stating the problem, the prompt contains much domain knowledge which can aid the search. However, competing methods may not have the same domain knowledge.

Many existing works use different prompts when comparing to other pipelines [20, 32, 34]. However, we find that even the same pipeline can give drastically different results given two different prompts. To illustrate, we run ShinkaEvolve three times on the circle packing problem, using its default prompt with expert hints (see Listing 1) and with a prompt containing minimal domain knowledge (Appendix E). Otherwise, all hyperparameters are kept fixed, including Shinka's default initial program. When using the default expert prompt, Shinka finds the same bound as Lange et al. [20] 3/3 times, whereas given the minimal domain-knowledge prompt this bound is found in only 1/3 runs. For additional ablations, see Appendix F.

Thus, to ensure fair comparisons, throughout the paper all methods are given the same minimal domain-knowledge prompts unless noted otherwise. Where relevant, pipelines are also given an uninformative initial program.

### 4.2 Results

Table 1: Best bounds found using code evolution and baselines. Best results are bolded, second best are underlined. AlphaEvolve results are excluded from comparisons but included for reference as it uses an unknown but likely much higher budget and more domain knowledge. Sequential conditioned sampling (SCS) matches or exceeds ShinkaEvolve on most problems, with IID random sampling (IID RS) also exhibiting competitive performance, occasionally outperforming other methods. Arrows denote whether higher or lower is better. # problems $\geq M$ is the number of problems for which a method matches or exceeds method $M$. More significant digits are used when discovered bounds are close. Each result is achieved using a $20 budget.

| | | | Baselines (ours) | |
|---|---|---|---|---|
| Problem | AlphaEvolve | ShinkaEvolve | IID RS | SCS |
| # problems $\geq$ ShinkaEvolve | 7/9 | | 4/9 | 6/9 |
| # problems $\geq$ AlphaEvolve | | 3/9 | 2/9 | 4/9 |
| Average rank (↓) | | 1.9 | 2.3 | 1.8 |
| First autocorr. ineq. (↓) | 1.505 | <u>1.522</u> | 1.535 | **1.519** |
| Second autocorr. ineq. (↑) | 0.8962 | **0.8955** | 0.8739 | 0.8795 |
| Uncertainty ineq. (↓) | 0.3521 | **0.3521** | **0.3521** | **0.3521** |
| Erdős' min. overlap (↓) | 0.3809 | **0.3810** | <u>0.3811</u> | 0.3812 |
| Sums/differences of sets (↑) | 1.1584 | 1.1095 | **1.1237** | <u>1.1178</u> |
| Max–min dist. ratio (↓) | 12.88926 | **12.88923** | **12.88923** | **12.88923** |
| Heilbronn triangles (↑) | 0.0365 | <u>0.0356</u> | 0.0334 | **0.0365** |
| Kissing number in 11D (↑) | 593 | <u>402</u> | **438** | **438** |
| Circle packing (↑) | 2.63586 | **2.63598** | 2.632 | <u>2.63590</u> |

Table 1 shows that both baselines perform well, matching or exceeding Shinka on $4/9$ and $6/9$ problems for IID RS and SCS respectively.[3] Interestingly, SCS matches or exceeds AlphaEvolve on $4/9$ problems as well, in spite of likely using a lower budget and less domain knowledge. Thus, the baselines seem similarly if not more performant than sophisticated code evolution methods.

To compare the different baselines' cost effectiveness, we calculate the probability that a method matches or exceeds ShinkaEvolve under a given budget. This is done using a bootstrap estimate, with details in Appendix G. Figure 2 demonstrates that both baselines compare favorably to ShinkaEvolve across all budgets. The probabilities initially decrease due to ShinkaEvolve having a warmup period, as it is based on file edits and not full-file generations. Relying on code diffs and other mechanisms, such as occasionally using cheaper LLMs, does not seem to make ShinkaEvolve significantly more cost-efficient.

Here a method's efficiency is measured relative to its API budget, whereas other constraints can be limiting as well. With respect to wall-clock time, the two baselines are much faster due to them being easily parallelizable across CPU cores, requiring 1-3 hours per problem relative to ShinkaEvolve's ∼10 hours. In Appendix I we find that the baselines are sample-efficient also with respect to the number of function evaluations.

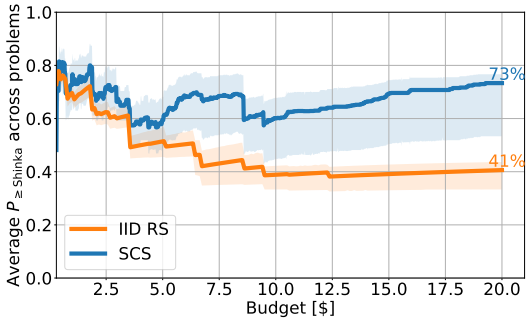



Figure 2: Average probability of matching or exceeding ShinkaEvolve at a given budget over the 9 problems for the two baselines. Colored numbers on the right are the probabilities at the max budget of $20. Both baselines perform well over different budgets, with sequential conditioned sampling (SCS) generally outperforming ShinkaEvolve. Shaded regions are asymmetric 95% confidence intervals, see Appendix G for details and Appendix H for per-problem breakdowns.

[3] Using all the baseline-generated programs, beyond the $20 limit, results in SCS matching ShinkaEvolve on circle packing and finding an improved bound of 1.1216 for sums/differences of sets. IID RS finds a slightly better bound of 1.529 for the first autocorrelation inequality.

### 4.3 A Problem's Search Space Dictates its Performance Ceiling

As shown in Table 1, all methods find a similar bound for some problems, while still finding similarly performing solutions for others. Thus, given a sufficient budget, all methods can likely reach the same performance ceiling. If this is the case then the only way to improve performance would be by expanding a problem's search space.

To test this, we take one of the problems and improve its formulation. Changing a problem's formulation is reflected by changing the code evolution's verifier and thus the problem's search space. Specifically, we take the uncertainty inequality and improve its formulation so it is easier to optimize and searches over a larger class of functions, with details in Appendix J.1. Note that AlphaEvolve's formulation is from Gonçalves et al. [12], with AlphaEvolve and all tested methods here improving the bound from 0.3523 to 0.3521. The new formulation results in all methods – both baselines and ShinkaEvolve – finding an improved bound of 0.3482 given the same $20 budget. Changing the formulation yields a larger improvement than search alone over a given setup, regardless of the search pipeline. We similarly analyze three additional problems in Appendix J.2.

Notably, a problem's formulation is designed by domain experts, not the code evolution systems. If good formulations are what make a problem amenable to automated search, with different search methods performing similarly, then many design choices in existing pipelines may be redundant.

The importance of a problem's search space is interesting in regards to *open-endedness*. Hu et al. [15], Xiong et al. [33] and Zhang et al. [35] discuss code evolution being theoretically Turing-complete and a form of open-ended search, but how open-ended are current code evolution systems if in practice they do not alter the most important parts of a problem? In this sense, current systems exhibit a confined form of open-endedness, whereas what matters for achieving better performance is altering parts of the setup which are currently kept fixed. Making code evolution more open-ended is an exciting avenue for future work.

## 5 Evolving Agentic Scaffolds is Highly Stochastic

To evaluate how baselines compare to code evolution on a problem where the main constraint is the number of function evaluations, we test how well different methods can design agentic scaffolds. In this setting, a method must design a series of LLM calls, called a scaffold, that solves some problem,

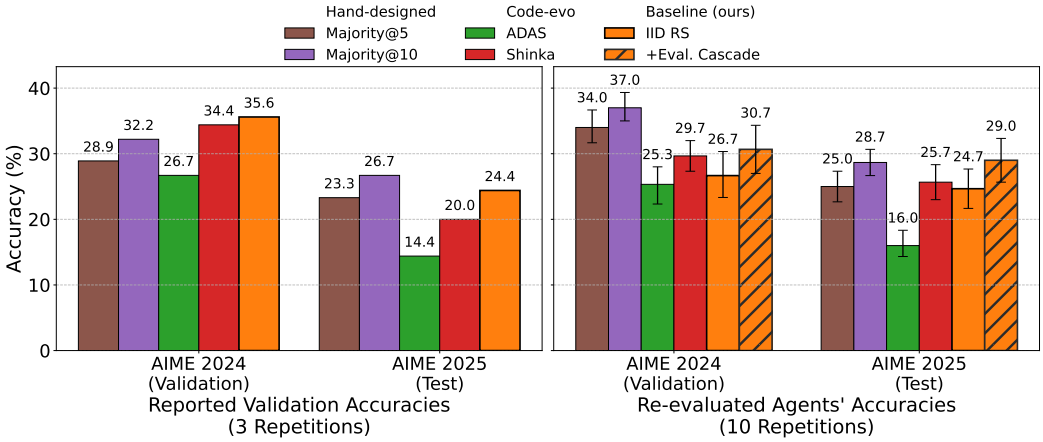



Figure 3: AIME 2024 and 2025 accuracies for different methods. Majority@5/10 indicate manually designed majority vote scaffolds. **(Left)** Validation accuracies are those measured while evolving different scaffolds, where both validation and test accuracies are based on 3 evaluations over the dataset. ShinkaEvolve numbers are from Lange et al. [20]. IID RS seemingly performs best out of the search methods, although all do worse than majority vote on the test set, with large drops in accuracy. **(Right)** Results when re-evaluating the scaffolds 10 times, with whiskers denoting 95% confidence intervals. Validation accuracies are lower for all automated search methods as their scaffolds are seemingly selected more so due to stochasticity in the evaluations than them achieving good performance. Unlike when evaluating only 3 times, it is apparent that there is no clear difference between ShinkaEvolve and IID RS, with the probability of improvement being $P(\text{Shinka} > \text{IID RS}) = 0.49$. Using an evaluation cascade with IID RS results in selecting a robustly higher-performing scaffold.

e.g. finding the correct answer to a math question. A meta-agent (the code evolution pipeline) designs a series of scaffolds, tests how well they do on some validation set, and then picks the best one and evaluates its performance on a test set [15, 20]. Typically, the limiting factor is not meta-agent queries but the number of evaluated scaffolds, as depending on which LLM is used each evaluation typically costs $1-10. For example, Hu et al. [15] use training runs of 30 generations, in which we estimate about 50 scaffolds are evaluated, to cost them $300.

We compare two code evolution methods, ADAS [15] and ShinkaEvolve, to IID RS. ADAS runs for 30 generations but can re-evaluate a scaffold to debug it, resulting in up to 90 evaluations. ShinkaEvolve runs for 75 but only about 60 generations result in an evaluated scaffold. Thus, we limit IID RS to evaluating 50 scaffolds, so a scaffold that does not compile does not count towards this limit. We test all methods' ability to design scaffolds for questions from the AIME math competition [1, 2], with all agents using GPT4.1-nano for their scaffold evaluations. Note that GPT4.1-nano's knowledge cutoff is June 2024 [24], before AIME 2025 was released. We compare the discovered scaffolds to a manually designed majority vote@$k$ scaffold. Majority vote means asking the model to answer the question $k$ times and then picking the most prevalent answer, with ties being resolved uniformly at random. Additional technical details are in Appendix K.

**Results.** Figure 3 (left) compares each method's scaffolds when evaluating them as done in some prior work, where validation accuracies used throughout the search are reported, with reported test accuracies being over similarly sized datasets. All code evolution methods, including the IID RS baseline, exhibit >10% drops in accuracy between their validation and test sets. Although it is likely that AIME 2025 is harder than 2024, this drop is only for automated search methods, with the majority vote baselines degrading by only about 5%.

Note that to keep the already costly evaluation economical, validation sets used to find scaffolds are typically constructed of ∼100 questions (including repetitions), regardless of the full dataset's size [see 15, Appendix E]. However, small datasets introduce high variance into the evaluations. To illustrate, Figure 4 shows the empirical AIME 2025 accuracy distribution when using a majority vote@5 scaffold, comparing the uncertainty when each question is evaluated a different number of times. Due to high variance, if scaffolds are evaluated on small datasets then they will be picked not due to their performance but mostly due to the evaluation's stochasticity, resulting in good validation performance being likely coincidental. Figure 3 (right) demonstrates this, as re-evaluating scaffolds discovered by automated methods 10 times shows that their underlying validation accuracies are lower than initially reported.

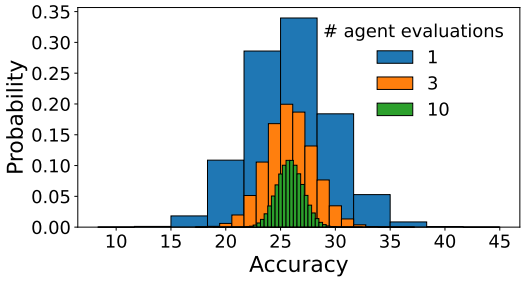



Figure 4: Empirical distributions of majority vote@5 accuracies for AIME 2025 when evaluating each question a different number of times. Even when evaluating 10 times there is a standard deviation of more than 1% in the accuracy. Distributions were calculated by sampling 100 answers to each question and bootstrapping.

### 5.1 Stochasticity Reduction

Reducing evaluation stochasticity by naïvely sampling more would be expensive. Taking AIME 2025 as a typical example, a standard deviation of 0.5% in the accuracy would require each question to be evaluated roughly $> 60$ times, which is $> 20\times$ as costly as the default evaluations used by ShinkaEvolve and ADAS. If only a scaffold's final performance is of interest then fewer samples are likely sufficient. Specifically, for reporting final results, we recommend methods to be evaluated over an effective test set size of at least 300 questions, equivalent to 10 repetitions here, and use 95% confidence intervals. Higher variance datasets may require more samples.

To reduce costs while also evolving scaffolds, we recommend using an *evaluation cascade*. As low performing scaffolds are not of interest, in a two-level cascade bad scaffolds can be evaluated over a small effective validation set while those that are potentially competitive are then re-evaluated over a larger set to see if their improvement holds. For IID RS this can easily be implemented by re-evaluating the top-X% of scaffolds using more repetitions. If the top-20% of scaffolds are re-evaluated then the evaluation cascade is estimated to be $2\times$ cheaper relative to naïvely using a larger validation set.

An additional source of noise in the evaluations is due to relying on summary statistics instead of comparing distributions. To illustrate, imagine a consistent and noisy scaffold, where the consistent one always gets an accuracy of 20% while the noisy scaffold has a 90% probability of getting an accuracy of 12% but 10% of the time an accuracy of 100%. Although in most cases the consistent scaffold outperforms the noisy one, the noisy scaffold has a higher mean accuracy.

To robustly compare two methods' test accuracies, we recommend using the *probability of improvement*, a metric for comparing two distributions. Introduced by Agarwal et al. [4] for comparing deep reinforcement learning methods, the probability of improvement measures the probability a single run of $A$ would outperform a single run of $B$, $P(A > B)$.

However, when evolving scaffolds or picking the best one, the comparisons are between sets of scaffolds and not pairs. To robustly compare sets of scaffolds, we introduce the *probability of dominance*, a generalization of the probability of improvement that tests whether a method is better than a set of alternatives. Explicitly, given methods (here, scaffolds) $A_1, A_2, ..., A_M$, the probability of dominance is the empirical estimate of $P(A_1 > A_2, A_3, ..., A_M)$, which is the probability a given run of $A_1$ would outperform ("dominate") a single run of each of $A_2, A_3, ..., A_M$. On both the probability of improvement and dominance see Appendix L for an extended discussion.

Using the probability of dominance with an evaluation cascade allows IID RS to find a better scaffold that robustly generalizes to the test set, see right plot in Figure 3. This scaffold outperforms all scaffolds found without robust evaluations and performs comparably well to majority vote@10.

## 6 Machine Learning Competitions

To see how the baselines compare to code evolution for tasks with limited wall-clock time, we test their performance on MLE bench, a benchmark of Kaggle competitions [6]. For each competition, each method is given 24 hours to produce a high-performing solution. All methods select their best-performing solution based on its validation accuracy. All runs are given a single RTX 8000 GPU with 12 CPU cores. We use the ten-competition subset of Schmidgall et al. [26] for evaluating the different methods.

We compare the baselines to AIDE, a code evolution pipeline designed specifically for Kaggle competitions [18]. Chan et al. [6] find that on MLE bench AIDE outperforms other code evolution systems, such as MLAB [16] and OpenHands [31]. Additional technical details are included in Appendix M.

**Results.** Table 2 shows each method's results for each competition. SCS and AIDE perform similarly well, with SCS beating it on 6/10 competitions. Simple baselines can thus perform well also in difficult time limited domains.

Table 2: Results for different methods on the Kaggle competitions. $*$, $+$, $\times$ denote scores that would have gotten bronze, silver, or gold medals respectively. Best and second-best method results per competition are respectively shown in bold and underlined. For unabbreviated competition names and human baseline metrics, see Appendix M.

| Competition | | AIDE | IID RS | SCS |
|---|---|---|---|---|
| # competitions $\geq$ AIDE | | | 2/10 | 6/10 |
| # competitions $\geq$ Median | | 8 | 4 | 8 |
| # competitions $\geq$ Bronze | | 5 | 3 | 5 |
| # competitions $\geq$ Silver | | 3 | 3 | 5 |
| # competitions $\geq$ Gold | | 2 | 2 | 2 |
| Insult Detection | (↑) | **0.9135**$^{\times}$ | <u>0.9056</u>$^{\times}$ | 0.8460$^{\times}$ |
| Dec. 2021 Tabular | (↑) | 0.9620$^{\times}$ | <u>0.9621</u>$^{\times}$ | **0.9623**$^{\times}$ |
| Trans. Conductors | (↓) | <u>0.0619</u>$^{+}$ | 0.0620$^{+}$ | **0.0616**$^{+}$ |
| English Text Norm. | (↑) | <u>0.9823</u> | 0.9332 | **0.9913**$^{+}$ |
| May 2022 Tabular | (↑) | **0.9965** | 0.9140 | <u>0.9839</u> |
| Random Pizza | (↑) | <u>0.6820</u> | 0.6206 | **0.6891** |
| Spooky Author | (↓) | **0.2883**$^{*}$ | 0.4468 | <u>0.4031</u> |
| Toxic Jigsaw | (↑) | <u>0.9807</u> | 0.9780 | **0.9863**$^{+}$ |
| Russ. Text Norm. | (↑) | **0.9756**$^{*}$ | <u>0.9735</u> | 0.9453 |
| NYC Taxi | (↓) | 11.460 | **5.1483** | <u>5.9758</u> |

## 7 Combinatorial Optimization

ALE-Bench [17] tests how well AI systems can design heuristics for difficult optimization problems, many likely being NP-hard. As the agentic scaffold evaluations proved highly stochastic, we use ALE-Bench LITE to test how the two baselines compare to complicated code evolution pipelines.

We benchmark different methods on the 10 ALE-Bench LITE tasks, generally following the setup used by Lange et al. [20]. Notably, instead of a special AWS instance we run problems locally and adjust per-problem timeouts accordingly so the initial program runs successfully. All methods are run on identical machines, allowing comparisons between them, but making comparisons with methods benchmarked using a different setup not be apples-to-apples. We report the performance of the best program per problem per method on the public test set, which corresponds to the top-1 evaluation of Lange et al. [20]. This evaluation has a low variance, so there is no need for an evaluation cascade or other more robust metrics.

All methods – ShinkaEvolve, IID RS, and SCS – generate 50 programs per problem. SCS has a single trial and 10 programs per generation. ShinkaEvolve uses the default hyperparameters used for the ALE-Bench experiments in Lange et al. [20], except with the non-Gemini models being disabled and using Gemini-2.5 Flash instead of GPT-5 mini as the meta-model.

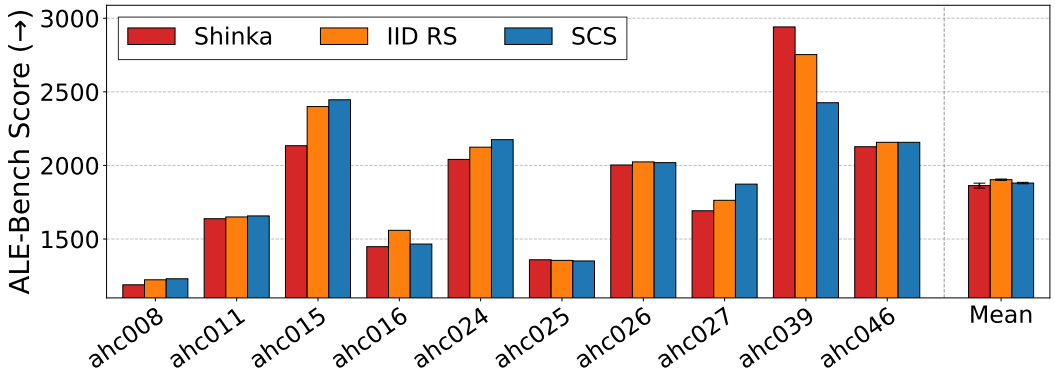



Figure 5: Relative performance of different methods on ALE-Bench LITE. IID RS and SCS outperform ShinkaEvolve on 8/10 problems and have a higher average score. Scores are the median of three evaluations of the best discovered program. Means are over the three evaluations, with uncertainties denoting standard errors.

Figure 5 shows the results. Both IID RS and SCS beat Shinka on 8/10 problems, with SCS' average performance being worse chiefly due to poor performance on problem ahc039, which may be an outlier.

Initially, we were surprised to see that both methods outperform ShinkaEvolve so clearly. There are a few possible reasons why this may be, one being that Shinka is inefficient given a finite sample budget due to sometimes doing file diffs instead of full file rewrites. Another theory is that ShinkaEvolve's strictly sequential structure can bias it down potentially suboptimal search paths, while both methods – especially IID RS – are relatively less biased. Whichever theory is correct, note that both support a similar point – that most design choices in existing complicated pipelines are likely redundant, if not detrimental.

## 8 Discussion

Across several domains, simple baselines are competitive with domain-specific sophisticated code evolution pipelines. This remains true across different constraints: a limited API budget when finding mathematical bounds, a limit on the number of function evaluations when designing agentic scaffolds or heuristics for combinatorial optimization, and limited wall-clock time in machine learning competitions. Accordingly, the main drivers for *improving* performance further come not from adding complexity to the search pipeline but parts of the setup external to the search pipeline, such as the search space, the domain knowledge, or the evaluation framework. Comparisons where part of the setup beyond the pipeline differs can lead to brittle conclusions on a method's performance.

These results likely hold also for much larger budgets. All budgets used here are either standard or above-standard for the relevant settings, e.g. with ShinkaEvolve using only $12 and not $20 for their circle packing experiments [20]. Additionally, after this paper was released, Tsoukalas et al. [30] noted that a very simple code evolution pipeline was competitive with AlphaEvolve in finding proofs for open mathematical theorems. Tsoukalas et al. [30] find similar results when using hundreds of dollars per experiment, beyond the tens of dollars used here.

Code evolution has achieved impressive capabilities across a range of domains, yet which design choices contribute to these successes has been largely overlooked. We find that much of the additional complexity in existing sophisticated pipelines does not translate to further improvements in performance. *Select* design choices may still be important for a specific domain but not others, see the need for textual feedback for MLE bench (Appendix M).

Why were the discovered shortcomings not identified previously? First, for many works in code evolution, different contributions may be conflated, leading baselines to often be ignored. For example, Novikov et al. [23] give two main contributions: they both introduce a search method, AlphaEvolve, and demonstrate various scientific discoveries and optimizations it achieved. Both are likely valuable in and of themselves, but achieving a scientific discovery does not necessarily imply that the method used to find it is performant.

Second, if not tightly controlling for fair comparisons, there can be apparent improvements which in practice stem not from the code evolution pipeline but something else, such as domain knowledge. Such accidental improvements can easily happen, as different published results are often used without rerunning exact setups, with matters like prompts potentially being seen as unimportant.

To facilitate better progress in the field, we recommend the following best practices for future work. **If proposing search methods, run fair comparisons and compare to simple baselines.** Comparisons should use the same language models, domain knowledge, search spaces, and so on. These are important not only for sharing robust results but for developing methods, as otherwise apparent improvements may not generalize across settings. The simple baselines and setup here provide a convenient test bed for when discrete changes to a pipeline are beneficial. **If showcasing a scientific discovery, disclose the domain knowledge and search setup used to find it.** The domain knowledge in prompts or initial programs, coupled with the search's structure, may take a larger part in enabling a discovery than the pipeline itself. Without disclosing exact setups the discovery might not be reproducible, nor allow being built upon. When evaluating noisy functions, such as when searching for agentic scaffolds, **use evaluation cascades and the probability of dominance to economically get robust results that better generalize.** Otherwise, scaffolds may underperform relative to simple existing setups. In general, the selected noisy discovered functions should be re-evaluated several times and with reported confidence intervals, to ensure reported results are robust.

To better enable rigorous code evolution development we include in Appendix N a set of tips to keep the development economical, allowing more runs to be used and different setups compared. Altogether, we hope these recommendations and findings will enable continued development of code evolution that matters.

## Acknowledgements

Thanks to much of the Sakana AI research team for fruitful discussions, and especially to Robert Lange for helping with the ShinkaEvolve runs and discussing the work throughout. Thanks as well to Yujin Tang, who provided good feedback and support that spurred the development of an earlier version of this work. Thanks also to Michael Beukman, Dulhan Jayalath, and Anushka Nair for feedback on an earlier draft, to Noya Gideoni and Katrina Dickson for proofreading it, and to Edan Toledo for feedback and some help with the MLE bench setup.

We thank the Oxford ARC cluster for providing the GPUs for the MLE bench experiments. Yonatan is funded by the Rhodes Trust and the AIMS EPSRC CDT (grant no. EP/S024050/1).

## References

[1] American invitational mathematics examination, 2024. Problems and solutions, 2024. Published by the Mathematical Association of America / AMC contests.

[2] American invitational mathematics examination, 2025. Problems and solutions, 2025. Published by the Mathematical Association of America / AMC contests.

[3] Julius Adebayo, Justin Gilmer, Michael Muelly, Ian Goodfellow, Moritz Hardt, and Been Kim. Sanity checks for saliency maps. *Advances in neural information processing systems*, 31, 2018.

[4] Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron C Courville, and Marc Bellemare. Deep reinforcement learning at the edge of the statistical precipice. *Advances in neural information processing systems*, 34:29304–29320, 2021.

[5] BenchLM. Llm api pricing comparison 2026. https://benchlm.ai/llm-pricing, 2026.

[6] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, et al. Mle-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[8] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[9] Batu El, Mert Yuksekgonul, and James Zou. Inefficiencies of meta agents for agent design. *arXiv preprint arXiv:2510.06711*, 2025.

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

[11] Alexander David Goldie, Zilin Wang, Jaron Cohen, Jakob Nicolaus Foerster, and Shimon Whiteson. How should we meta-learn reinforcement learning algorithms? *arXiv preprint arXiv:2507.17668*, 2025.

[12] Felipe Gonçalves, Diogo Oliveira e Silva, and Stefan Steinerberger. Hermite polynomials, linear flows on the torus, and an uncertainty principle for roots. *Journal of Mathematical Analysis and Applications*, 451(2):678–711, 2017.

[13] Ryan Greenblatt. Getting 50%(sota) on arc-agi with gpt-4o. *Redwood Research Blog, June*, 2024.

[14] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[15] Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint arXiv:2408.08435*, 2024.

[16] Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2023.

[17] Yuki Imajuku, Kohki Horie, Yoichi Iwata, Kensho Aoki, Naohiro Takahashi, and Takuya Akiba. Ale-bench: A benchmark for long-horizon objective-driven algorithm engineering. *Advances in Neural Information Processing Systems*, 38, 2026.

[18] Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. Aide: Ai-driven exploration in the space of code. *arXiv preprint arXiv:2502.13138*, 2025.

[19] Robert E Keller and Wolfgang Banzhaf. The evolution of genetic code in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1077–1082. Morgan Kaufmann Orlando, 1999.

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

[21] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

[22] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. *arXiv preprint arXiv:2310.12931*, 2023.

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

[24] OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, April 2025.

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

[26] Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using llm agents as research assistants. *arXiv preprint arXiv:2501.04227*, 2025.

[27] Rulin Shao, Shuyue Stella Li, Rui Xin, Scott Geng, Yiping Wang, Sewoong Oh, Simon Shaolei Du, Nathan Lambert, Sewon Min, Ranjay Krishna, et al. Spurious rewards: Rethinking training signals in rlvr. *arXiv preprint arXiv:2506.10947*, 2025.

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

[29] Edan Toledo, Karen Hambardzumyan, Martin Josifoski, Rishi Hazra, Nicolas Baldwin, Alexis Audran-Reiss, Michael Kuchnik, Despoina Magka, Minqi Jiang, Alisia Maria Lupidi, et al. Ai research agents for machine learning: Search, exploration, and generalization in mle-bench. *arXiv preprint arXiv:2507.02554*, 2025.

[30] George Tsoukalas, Anton Kovsharov, Sergey Shirobokov, Anja Surina, Moritz Firsching, Gergely Bérczi, Francisco JR Ruiz, Arun Suggala, Adam Zsolt Wagner, Eric Wieser, et al. Advancing mathematics research with ai-driven formal proof search. *arXiv preprint arXiv:2605.22763*, 2026.

[31] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

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

[33] Yiming Xiong, Shengran Hu, and Jeff Clune. Learning to continually learn via meta-learning agentic memory designs. *arXiv preprint arXiv:2602.07755*, 2026.

[34] Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, et al. Learning to discover at test time. *arXiv preprint arXiv:2601.16175*, 2026.

[35] Jenny Zhang, Bingchen Zhao, Wannan Yang, Jakob Foerster, Jeff Clune, Minqi Jiang, Sam Devlin, and Tatiana Shavrina. Hyperagents. *arXiv preprint arXiv:2603.19461*, 2026.

# A Related work

While modern code evolution relies on LLMs, code evolution as a term has existed at least as far back as 1999 [19]. Although only an emerging field, there are a plethora of works on code evolution, we focus here on some notable representatives. Some earlier LLM-based code evolution methods have components similar to the two given baselines. FunSearch [25] and AlphaCode [21] have components similar to IID RS.

There are also works that implement pipelines similar to SCS, often with domain-specific adjustments. For example, Ma et al. [22] use an SCS-like code evolution pipeline to find better reward functions for reinforcement learning environments.

In other subfields, especially reinforcement learning, there have been many efforts to improve the field's robustness. Henderson et al. [14] discuss difficulties in reproducing existing work in RL while Agarwal et al. [4] showcase problems in evaluations, proposing various best practices. Goldie et al. [11] compare a variety of meta-RL algorithms, discussing best practices for future work.

Regarding agentic scaffolds, El et al. [9] discuss inefficiencies in searching for agentic scaffolds. They note that typical scaffolds are expensive to run when measured based on the price per correct answer, but do not compare to different search methods or propose methods to make the search cheaper.

# B AlphaEvolve Problems

Table 3 lists the 9 problems used in §4, taken from Novikov et al. [23].

Table 3: Bounds of AlphaEvolve (AE) problems studied here, divided into analysis (top), combinatorics (middle), and geometry (bottom). The arrow next to the problem name indicates whether it is an upper bound, so lower results are tighter and hence better (↓), or a lower bound so higher is tighter and thus better (↑). All numbers are from Novikov et al. [23]. "Pre-AE" are the best bounds from before Novikov et al. [23].

| Problem | Input size | Pre-AE Bound | AE Bound | AE Appendix |
|---|---|---|---|---|
| First autocorrelation inequality (↓) | Unbounded, step function heights | 1.5098 | 1.5053 | B.1 |
| Second autocorrelation inequality (↑) | Unbounded, step function heights | 0.88922 | 0.8962 | B.2 |
| Uncertainty inequality (↓) | 3 coefficients of a Hermite polynomial | 0.3523 | 0.3521 | B.4 |
| Erdős' minimum overlap (↓) | Unbounded, step function heights | 0.380927 | 0.380924 | B.5 |
| Sums vs. differences of finite sets (↑) | Unbounded, set $U \subset \mathbb{Z}_{\geq 0}$ fulfilling some properties | 1.14465 | 1.1584 | B.6 |
| Max–min distance ratio for 16 2D points (↓) | 32 coordinates (16×2) | 12.890 | 12.88927 | B.8 |
| Heilbronn triangles $n = 11$ (↑) | 22 coordinates (11×2) in a unit-area triangle | 0.036 | 0.0365 | B.9 |
| Kissing number in 11D (↑) | Largest number of 11D sphere centers all tangent to a common sphere | 592 | 593 | B.11 |
| Circle packing (↑) | 78 – 26 center coordinates (26×2) and 26 radii | 2.634 | 2.63586 | B.12 |

## C Finding Bounds for Math Problems: Technical Details

For the baselines we use Gemini 2.5 Pro, sampling with a temperature of $0.8$, a top-$p$ sampling cutoff of $0.95$, a thinking budget of 1024 tokens, and let each program evaluation run for at most 5 minutes. These settings were not thoroughly tuned and were chosen as they seemed like sensible defaults. IID RS samples 2000 programs and SCS samples 20 programs per generation, having 10 generations per trial and 6 trials overall, resulting in 1200 generated programs. This is where the numbers in Figures 1b and 1c come from. After the first generation in each trial, SCS randomly picks 3 programs that successfully ran in the previous generation and appends them and their bounds to the IID RS prompt when generating a new program.

For fairness, ShinkaEvolve had some of its hyperparameters slightly tuned, as at its default settings it was not competitive with the baselines. Most importantly, it also uses a thinking budget of 1024 tokens, as otherwise it is much more expensive and results in far fewer generations. Other hyperparameters are taken from ShinkaEvolve's default circle packing configuration, except for using 5 island subpopulations instead of 2 due to the slightly larger budget of $20 instead of the default setup's $12, and uses 2 programs from its archive for inspiration instead of the default 4 in order to reduce API costs.

## D OpenEvolve Results on Math Problems

When running OpenEvolve [28] using Gemini models we found it to often crash due to getting `None` results, while also not having a way to specify LLM thinking budgets or a max API budget for a run. Table 4 shows its results given 300 generations when using GPT-5.2 and GPT-5 mini for the problems it managed to run successfully. For the six problems it ran successfully, OpenEvolve got an average subpar rank of 3.08 relative to the baselines and ShinkaEvolve.

Table 4: Best bounds found by OpenEvolve given 300 generations using GPT-5.2 and GPT-5 mini. "–" denotes a failed run. Arrows denote whether higher or lower is better.

| Problem | | OpenEvolve |
|---|---|---|
| First autocorr. ineq. | (↓) | – |
| Second autocorr. ineq. | (↑) | 0.8903 |
| Uncertainty ineq. | (↓) | 0.3521 |
| Erdős' min. overlap | (↓) | – |
| Sums/differences of sets | (↑) | 1.1095 |
| Max–min dist. ratio | (↓) | – |
| Heilbronn triangles | (↑) | 0.0354 |
| Kissing number in 11D | (↑) | 348 |
| Circle packing | (↑) | 2.541 |

## E Minimal Domain-Knowledge Circle Packing Prompt

The `verify_circles` function is taken from AlphaEvolve's validation script. `${max_execution_time}` is replaced with the time limit per problem, which in practice was 300 seconds (5 minutes) for the programs in Table 1. The minimal domain knowledge prompt used for all tested methods in Table 1 is:

```
You are an expert programmer specialising in numerical optimisation. Implement a Python function with the
↪  exact signature:

def pack_circles() -> Tuple[np.ndarray, np.ndarray, float]:

The function must pack 26 non-overlapping circles into the unit square [0,1]x[0,1] so that the sum of
↪  their radii is maximised. Returns:
- centers: np.ndarray of shape (26, 2) with (x, y) coordinates
- radii:   np.ndarray of shape (26,) with positive radii
- sum_radii: float = radii.sum()

You can use these predefined helper functions without redefining them:
```
```
```

```
import numpy as np
import itertools

def verify_circles(circles: np.ndarray) -> bool:
    """Checks that the circles are disjoint and lie inside a unit square.

    Args:
      circles: A numpy array of shape (num_circles, 3), where each row is
        of the form (x, y, radius), specifying a circle.

    Returns:
      True if all circles are disjoint and fully inside the unit square,
      False otherwise.
    """
    # Check pairwise disjointness.
    for circle1, circle2 in itertools.combinations(circles, 2):
        center_distance = np.sqrt((circle1[0] - circle2[0])**2 + (circle1[1] - circle2[1])**2)
        radii_sum = circle1[2] + circle2[2]
        if center_distance < radii_sum:  # Overlap
            return False

    # Check all circles lie inside the unit square [0,1]x[0,1].
    for circle in circles:
        x, y, r = circle
        if x - r < 0 or y - r < 0 or x + r > 1 or y + r > 1:
            return False

    return True
```

All circles must be fully inside the square and not overlap. You have up to ${max_execution_time} seconds for your solution to run. Please only supply the code for pack_circles, please define helper functions inside it.

## F Domain Knowledge Ablations

To see whether simple methods can leverage domain knowledge similarly to complicated code evolution pipelines, and whether prompts with domain knowledge transfer between methods, we run several ablations. First, in the workshop version of this work we found that giving IID RS a better prompt and 1000 samples led it to find the same "optimal" circle packing bound that ShinkaEvolve finds. Specifically, the prompt included a compute max radii function akin to the one that ShinkaEvolve has by default, but one which is exact and not a heuristic. Thus, simple methods can leverage domain knowledge as well.

Note that ShinkaEvolve's expert prompt was designed specifically for ShinkaEvolve, and likely specifically for the kind of errors it makes. We speculate it was designed by running ShinkaEvolve, seeing which errors it makes, and then adding instructions in the prompt to mitigate those errors. This is further supported by seeing that it finds the same "optimal" circle packing only 1/3 times when not given this domain knowledge, see section 4.1. Thus, if method 1 makes type A errors 50% of the time and type B errors the other 50%, while method 2 makes type A errors 30%, type B 20%, and type C 50%, a prompt saying "don't do A" would help method 1 much more than method 2, and for method 2 might even be detrimental.

This is what we observe in practice. To test whether ShinkaEvolve's prompt transfers to other methods, we test how well the two baselines perform given ShinkaEvolve's circle packing prompt, see Listing 1. Given ShinkaEvolve's "expert" prompt, IID RS and SCS find circle packings with a sum of radii of 2.626 and 2.621 respectively, both worse than when they are given no domain knowledge at all. These results imply domain knowledge, depending on how it is used, can be detrimental to one method while being beneficial to another.

## G Bootstrap Estimate

**IID Random Sampling.** Here the bootstrap estimate amounts to calculating a pass@k. Under a given budget we find the last generation ShinkaEvolve reached before exceeding it, and as its score take the best bound found until then. If the budget allows sampling $k$ IID programs then the probability of sampling one that matches or outperforms Shinka's score can be approximated by repeatedly sampling $k$ programs out of the $n$ total generated, here being 2000, and seeing if any match or exceed Shinka's score. If $c$ is the number of programs that match/exceed code evolution's

score this probability estimate can be analytically calculated as $1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$, as per Chen et al. [7]. The average IID cost per program is estimated as the average cost over all sampled programs, so small cost differences between individual generations are ignored.

**Sequential Conditioned Sampling.** For the SCS bootstrap estimate it is important to consider a realistic scheme of how a given budget would be exhausted, with different setups potentially yielding different results. Here it is assumed that a budget is used up, generation by generation, until a trial is exhausted, after which a new trial is begun. The bootstrap is estimated by picking a random trial, seeing up until which generation in it can be searched under the given budget, and if the trial is exhausted then continuing to one of the remaining trials. This is how the results in Table 1 were picked, except after exhausting a trial the next was picked not at random but chronologically.

Formally, if $p_{\geq \text{Shinka}}(B, T)$ is the probability of matching/exceeding a bound $s$ given a budget $B$ and a set of trials $T$, where trial $t \in T$'s budget is $b_t$, then $p_{\geq \text{Shinka}}$ is recursively defined as

$$p_{\geq \text{Shinka}}(B, T) = \frac{1}{|T|} \sum_{t \in T} \max(\mathbb{1}_{\max(t) \geq s}, p_{\geq \text{Shinka}}(B - b_t, T \setminus \{t\})) \tag{1}$$

This assumes that all $T$ trials are within the budget, where in practice some might not be while others are. In these cases the sum is only over the trials within budget. For minimization problems the $\max(t) \geq$ should be changed to $\min(t) \leq$.

**Confidence Intervals.** In both cases confidence intervals are calculated by bootstrapping over the bootstrap, i.e. picking programs/trials with replacement, calculating their probabilities of matching/exceeding, and then calculating the 2.5% and 97.5% quantiles over this empirical distribution over the matching/exceeding probabilities. Note that the uncertainties, e.g. in Figure 6, are large as there is often a >2.5% probability of getting a worse answer than ShinkaEvolve. However, these estimates are likely overly negative as there exists counterfactual information for worse answers – other sampled programs – but not for getting a better answer. This coupled with the probability of matching/exceeding's calculation being nonlinear is why the uncertainties are asymmetric and tend to have low lower bounds.

# H Figure 2 Per-problem Breakdown

# I Math Bounds Sample Efficiency

Figures 7 and 8 show respectively the aggregate and per-problem probability of the baselines matching/exceeding Shinka when the budget is defined not by the API cost but the number of evaluated programs. ShinkaEvolve evaluates one program per generation while the baselines evaluate one program per sample. Note that for a set API budget Shinka runs until different numbers of generations, as the average cost per program differs per problem.

Shinka being a bit cheaper per program, perhaps due to also using some cheaper LLMs, results in the final probabilities in Figure 7 being lower than those in Figure 2, but only slightly.

# J Different Problem Formulations

## J.1 Improved Uncertainty Inequality Formulation

We first describe the problem in its generality, based on Appendix B.4 of Novikov et al. [23] and Gonçalves et al. [12, p. 679]. For a function $f : \mathbb{R} \to \mathbb{R}$ define its Fourier transform as $\hat{f}(x) = \int_{-\infty}^{\infty} f(t)e^{-2\pi ixt}dt$. Let the radius of the smallest disc for which outside of it $f$ is nonnegative be defined as $A(f) \coloneqq \inf(\{r > 0 | \forall |x| \geq r : f(x) \geq 0\})$. In the uncertainty inequality problem we wish to find the largest constant $C$ for which $A(f)A(\hat{f}) \geq C$, under the conditions where a) $f$ is even and b) $\max(f(0), \hat{f}(0)) \leq 0$.[4]

[4]For this last condition Novikov et al. [23] mistakenly state it as $< 0$, see Gonçalves et al. [12, p. 679].

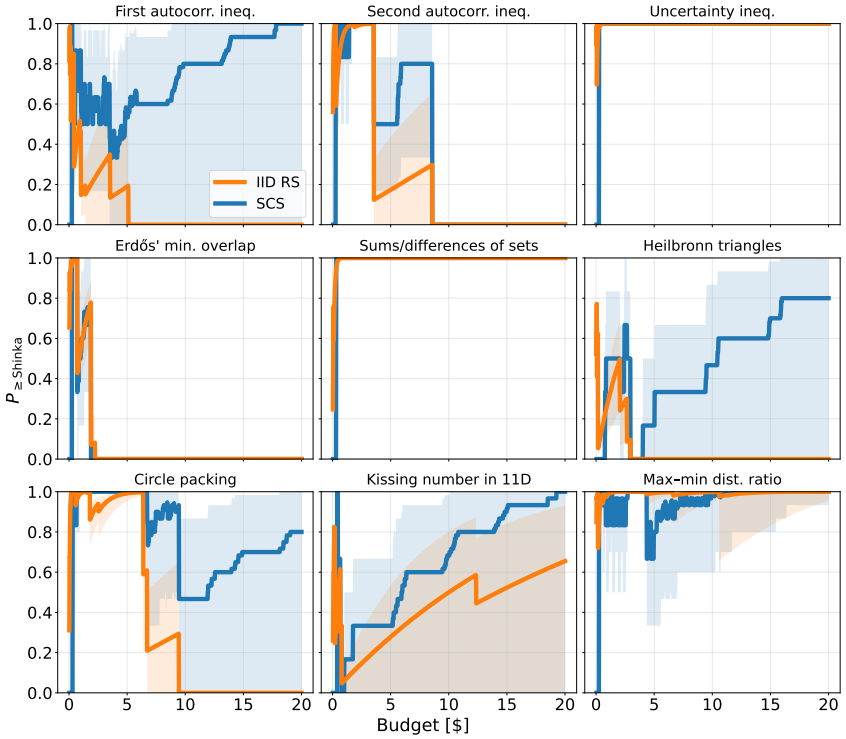



Figure 6: Per-problem probability of matching/exceeding ShinkaEvolve for the two baselines. Shaded regions are 95% confidence intervals.

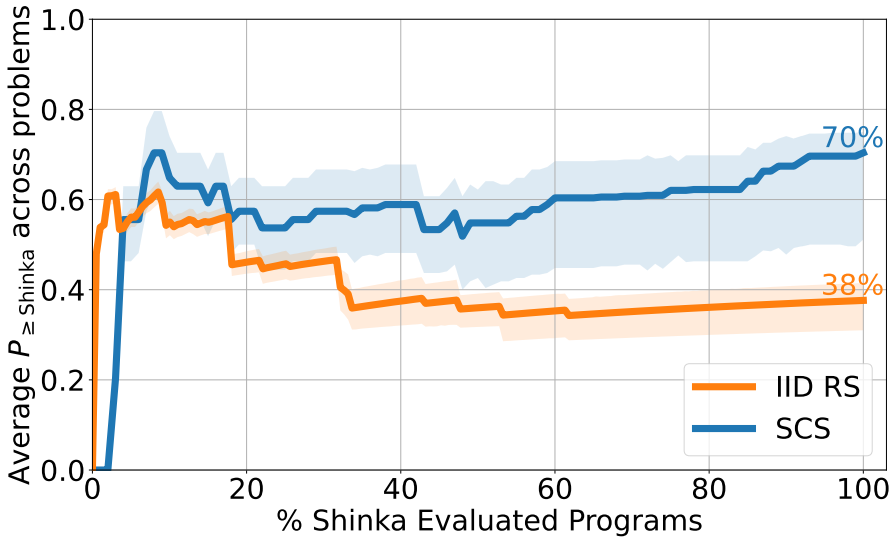



Figure 7: For each baseline, the average probability of matching or exceeding ShinkaEvolve across the 9 problems, as a function of the number of evaluated programs. The $x$ axis is the percent of evaluated programs out of the maximum in the corresponding ShinkaEvolve run. Shaded regions are 95% confidence intervals.

Denoting the $n$th Hermite polynomial as $H_n$, Gonçalves et al. [12, p. 697] show that functions of the form $f(x) = \sum_{n=0}^{\infty} \alpha_n H_{4n}(\sqrt{2\pi}x)e^{-\pi x^2}$ fulfill the two conditions given that the coefficients $\alpha_n$ are chosen so $f(0) = 0$. As here $\hat{f}(x) = f(x)$, this automatically fulfills condition b). As even Hermite functions are even, this fulfills condition a).

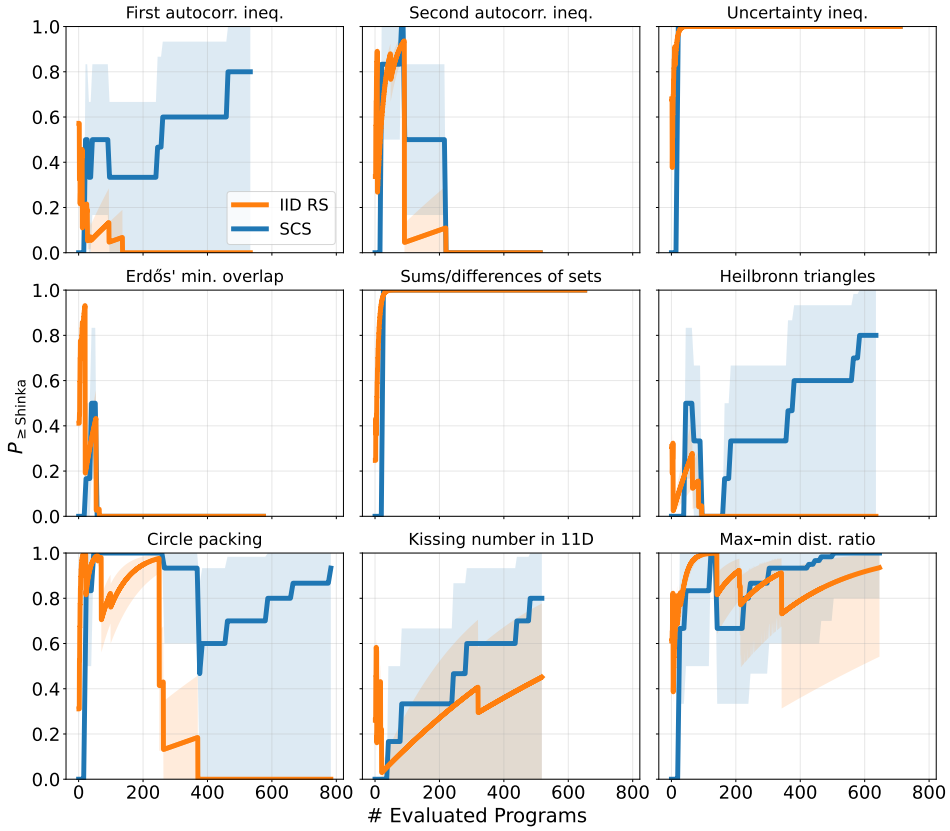



Figure 8: For each baseline, the per-problem probability of matching or exceeding ShinkaEvolve across the 9 problems, as a function of the number of evaluated programs. Shaded regions are 95% confidence intervals.

Gonçalves et al. [12] construct their upper bound of 0.3523 by setting all $\alpha_n$ except for $\alpha_0, \alpha_1, \alpha_2, \alpha_3$ to zero and numerically finding which $\alpha$s minimize $C$. This is the formulation also used by Novikov et al. [23] and in our Table 1.

We modify this formulation in two ways. First, Novikov et al. [23] use physicist's Hermite polynomials, where the leading coefficient of $H_n$ is $2^n$. This leads to numerical instabilities when attempting to use higher orders. Instead, we use the probabilist's Hermite polynomials, which are the same but rescaled, so $He_n := \frac{H_n}{2^n}$, resulting in the leading coefficient for all polynomials being one. Our second modification is setting all $\alpha$s beyond $\alpha_7$ to zero instead of $\alpha_3$. This allows $f(x)$ to represent a larger class of functions. It is likely possible going beyond $\alpha_7$ and reducing the bound further but we encountered numerical instabilities when trying to do so, likely from using very high order polynomials.

### J.2 Different Formulations for Additional Problems

It is simple to illustrate that larger search spaces lead to better performance by showing that existing methods perform worse when a problem's search space is artificially restricted. We demonstrate this over three problems, where in all of them all tested methods perform similarly well.

For circle packing, when all methods are limited to using circles with a maximum radius of 0.1, ShinkaEvolve and the two baselines all find the same bound of 2.514. For reference, the maximum radius found in good solutions for the unconstrained case is about 0.13, with those solutions achieving a bound of about 2.64.

For the second autocorrelation inequality, when the constructed step function is limited to use only 10 steps instead of an arbitrary amount, ShinkaEvolve finds a bound of 0.868, while the baselines find a new bound of 0.867. For reference, the AlphaEvolve bound is ∼0.896.

For the Erdős' minimum overlap problem, when also restricted to 10 steps all methods find a worse bound of 0.3824, up from the known best of 0.3809.

## K Agentic Scaffold Evolution Technical Details

Lange et al. [20] use Gemini-2.5 Pro, GPT-o4 mini, and Claude 4 Sonnet for ShinkaEvolve's meta-agent when designing scaffolds. IID RS here uses their prompt and with Gemini-2.5 Pro, without a limit on the thinking budget as the cost for designing a scaffold is almost negligible relative to the evaluation costs. ADAS is run using GPT-4o and by manually changing the ADAS prompt for the GPQA dataset to fit AIME's format, as ADAS has specialized prompts to fit its general setup. We attempted running ADAS with Gemini-2.5 Pro as its meta-agent but found it to perform worse, often yielding pipelines that fail due to creating plots or trying to evaluate code its evaluation agent would produce. For all methods the scaffold was limited to use up to 10 LLM calls per question.

Following ShinkaEvolve, each scaffold is evaluated on AIME 2024 three times. Each year the AIME competition consists of 30 questions, so this results in 90 question evaluations overall.

## L Probabilities of Improvement and Dominance

**Probability of Improvement.** Given methods $A, B$ that get scores $a_1, ..., a_N$ and $b_1, ..., b_N$ on some benchmark, following Agarwal et al. [4] the probability of improvement is

$$P(A > B) = \frac{1}{N}\sum_{i=1}^{N} S(a_i, b_i) \quad \text{where} \quad S(a,b) = \begin{cases} 1, & \text{if } a > b, \\ \frac{1}{2}, & \text{if } a = b, \\ 0, & \text{if } a < b. \end{cases} \tag{2}$$

Note that uncertainty over $P(A > B)$ can also be calculated using a bootstrap estimate. In cases where the probability of matching or exceeding is of interest, as in some parts of this work, the $\frac{1}{2}$ for when $a = b$ can be replaced with a 1. See Agarwal et al. [4] for additional discussions.

**Probability of Dominance.** The probability of dominance is the probability method $A_1$ is better than ("dominates") methods $A_2, A_3, ..., A_M$. We denote their scores as $a_1^{(1)}, a_2^{(1)}, ..., a_N^{(1)}, a_1^{(2)}, a_2^{(2)}, ...$, with the upper index indicating the method. $P(A_1 > A_2, A_3, \ldots, A_M)$ is defined as

$$P(A_1 > A_2, A_3, \ldots, A_M) = \frac{1}{N^M} \sum_{a^{(1)} \in A_1} \cdots \sum_{a^{(M)} \in A_M} S\Big(a^{(1)}, \ldots, a^{(M)}\Big), \tag{3}$$

where

$$S(a^{(1)}, \ldots, a^{(M)}) = \begin{cases} 1, & \text{if } a^{(1)} > \max_{m \geq 2} a^{(m)}, \\ \dfrac{1}{\left|\{m \geq 2 : a^{(m)} = a^{(1)}\}\right|}, & \text{if } a^{(1)} = \max_{m \geq 2} a^{(m)}, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The second case in $S$ means that success probabilities are evenly split across the top methods in the case of ties. We refrain from the notation $P(A_1 > \max(A_2, ..., A_M))$ as the probabilities are calculated over the empirical distributions, not summary statistics. Although illustrated here for $A_1$ versus $A_2, ..., A_M$ note that the method ordering is arbitrary.

Although it is expensive to calculate the probability of dominance exactly, it can be efficiently estimated using Monte-Carlo. Comparing more methods will lead any individual method's probability of dominance to generally be lower, with several top methods likely having similar probabilities.

**Using Probability of Dominance with Evaluation Cascades.** We refrain from giving exact prescriptions as to the number of questions and cascades as what is sufficient is likely case dependent, noting that 300 questions is a sensible minimum for an expensive evaluation. As the effects of stochasticity worsen as more scaffolds are evaluated, stricter cascades should be used, using independent reruns to ensure an agent's performance is not spurious.

# M MLE Bench Technical Details

We use AIDE prompts for the baselines, removing references to memory (previous programs) and using a single trial for SCS due to the time limit. All methods use Gemini-2.5 Pro and rely on the AIDE and MLE bench setups given in Toledo et al. [29].

Notably, due to the competitions' intricacy, zero-shot generated programs almost never run without errors. This is due to there being many minute details in the dataset or the environment that might not be mentioned in the competition's description, e.g. very few samples in some classes leading models to fail when expecting at least a certain number of data points. Thus, we add a step to both baselines, where after generating a solution they iteratively debug it until it runs successfully. This results in slightly more complicated baselines and demonstrates a domain in which textual feedback is evidently important for functional code evolution, unlike the previous two domains. Unlike AIDE, the baselines do not include explicit fitness-based selection or a memory of all past programs and their performance.

## M.1 Full MLE Bench Competition Names

Table 5 lists the unabbreviated names of the competitions in Table 2.

Table 5: Competition abbreviations and full names.

| Abbreviation | Full Competition Name |
|---|---|
| Insult Detection | Detecting Insults in Social Commentary |
| Dec. 2021 Tabular | Tabular Playground Series – December 2021 |
| Trans. Conductors | Nomad2018: Predict Transparent Conductors |
| English Text Norm. | Text Normalization Challenge – English Language |
| May 2022 Tabular | Tabular Playground Series – May 2022 |
| Random Pizza | Random Acts of Pizza |
| Spooky Author | Spooky Author Identification |
| Toxic Jigsaw | Jigsaw Toxic Comment Classification Challenge |
| Russ. Text Norm. | Text Normalization Challenge – Russian Language |
| NYC Taxi | New York City Taxi Fare Prediction |

## M.2 Human Baseline Metrics

Table 6 shows the human baseline metrics used to calculate reference medal and median scores in Table 2.

Table 6: Human baseline metrics on the Kaggle competitions. Median is the median score in the competition, with bronze, silver, and gold denoting the cutoff score to receive each medal. Metrics are from Schmidgall et al. [26].

| | | Human baseline metrics | | | |
|---|---|---|---|---|---|
| Competition | | Median | Bronze* | Silver+ | Gold× |
| Insult Detection | (↑) | 0.778 | 0.791 | 0.823 | 0.833 |
| Dec. 2021 Tabular | (↑) | 0.953 | 0.956 | 0.956 | 0.956 |
| Trans. Conductors | (↓) | 0.069 | 0.065 | 0.062 | 0.055 |
| English Text Norm. | (↑) | 0.990 | 0.990 | 0.991 | 0.997 |
| May 2022 Tabular | (↑) | 0.972 | 0.998 | 0.998 | 0.998 |
| Random Pizza | (↑) | 0.599 | 0.692 | 0.724 | 0.979 |
| Spooky Author | (↓) | 0.418 | 0.293 | 0.269 | 0.165 |
| Toxic Jigsaw | (↑) | 0.980 | 0.986 | 0.986 | 0.987 |
| Russ. Text Norm. | (↑) | 0.975 | 0.975 | 0.982 | 0.990 |
| NYC Taxi | (↓) | 3.597 | 2.923 | 2.881 | 2.337 |

## N Tips for Economic Code Evolution Development

Many of the experiments in this paper are expensive. To enable easier development of code evolution methods we discuss a few tips to help keep costs low.

First, when developing, one could use a cheaper LLM. For example, for diagnostics we used Gemini-2.5 Flash Lite or Flash instead of Pro and sampled fewer programs. Another option is to use open-source LLMs which allow self-hosting. E.g. Toledo et al. [29] use a Deepseek model instead of API calls, albeit chiefly to have a higher throughput due to API rate limits. However, open-source models have the clear downside of being generally less capable than various closed source counterparts, and require renting GPUs if none are available. Still, capable open-source models are estimated to cost $5 - 15\times$ less than frontier closed-source models [5].

Second, there are ways to make expensive evaluations cheaper. The evaluation cascade described in §5.1 helps keep the cost of agent evaluations low while still reducing stochasticity. Schmidgall et al. [26] run MLE bench experiments on laptop CPUs, thereby requiring only API calls and no GPUs. Although this limits their available toolset, it still allows comparing different pipelines under identical settings.

## O Additional Limitations

Magnitudes of improvements are not compared for the mathematical bounds and machine learning competitions' results as small magnitudes can be meaningful, making it unclear how to effectively compare them. For example, the difference between the SCS and ShinkaEvolve circle packing bounds in Table 1 is on the order of $10^{-4}$. This difference is meaningful when it is close to the best bound found but generally meaningless otherwise. Instead, we opt to use pairwise comparisons as they can be meaningfully interpreted, with the downside of results over individual tasks being less meaningful.

Specifically for the MLE bench experiments, we note that although AIDE is more complicated than the baselines, it is still simpler than some other code evolution systems. We are unaware of a code evolution method that is both more complicated and more performant on MLE bench, and compare to AIDE due to it being both well known and performant.

## NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: All claims made in the abstract and introduction are evidenced throughout the paper, see the last two paragraphs of the introduction for references.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: See the second paragraph in the discussion and Appendix O.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: The sole theoretical result is that the reformulation of the uncertainty inequality is still mathematically valid, see Appendix J.1 for details.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The paper discloses all setups, giving additional technical details throughout and in the Appendix. Code is released to facilitate reproducibility.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: See supplementary material.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: This is detailed throughout the paper and in the Appendix.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: See Figures 2, 3, and the discussion on proper benchmarking in §5.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)

- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: This is detailed throughout the main paper and Appendix. We specifically state the main budget constraint (API costs, function evaluations, or wall-clock time) and the setup used to run all experiments.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research conforms with the code of ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: This paper discusses foundational work in code evolution, with no direct societal impacts beyond those generally driven by the subfield.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper poses foundational work in code evolution and does not clearly pose any such risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All original works are credited and cited.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: See accompanying code in the supplemental material and README therein.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper includes no crowdsourcing or experiments with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper includes no crowdsourcing or experiments with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: The paper heavily relies on LLM usage for the various methods and baselines and makes this usage clear throughout.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.