SIMPLE BASELINES ARE COMPETITIVE WITH CODE EVOLUTION

Yonatan Gideoni[†*] **Sebastian Risi**[§‖] **Yarin Gal**[†]
[†]University of Oxford [§]Sakana AI [‖] IT University of Copenhagen

ABSTRACT

Code evolution is a family of techniques that rely on large language models to search through possible computer programs by evolving or mutating existing code. Many proposed code evolution pipelines show impressive performance but are often not compared to simpler baselines. We test how well two simple baselines do over three domains: finding better mathematical bounds, designing agentic scaffolds, and machine learning competitions. We find that simple baselines match or exceed much more sophisticated methods in all three. By analyzing these results we find various shortcomings in how code evolution is both developed and used. For the mathematical bounds, a problem's search space and domain knowledge in the prompt are chiefly what dictate a search's performance ceiling and efficiency, with the code evolution pipeline being secondary. Thus, the primary challenge in finding improved bounds is designing good search spaces, which is done by domain experts, and not the search itself. When designing agentic scaffolds we find that high variance in the scaffolds coupled with small datasets leads to suboptimal scaffolds being selected, resulting in hand-designed majority vote scaffolds performing best. We propose better evaluation methods that reduce evaluation stochasticity while keeping the code evolution economically feasible. We finish with a discussion of avenues and best practices to enable more rigorous code evolution in future work.

1 INTRODUCTION

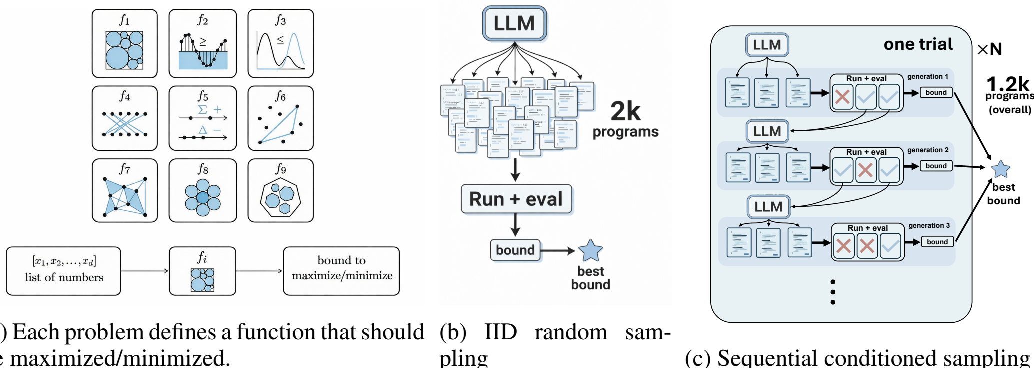
For many problems, a solution can take the form of a computer program, thereby allowing automated program search to solve a variety of tasks. This has been demonstrated across several domains: for scientific discovery, Novikov et al. (2025) find programs that improve on several mathematical bounds. Hu et al. (2024) use LLMs to design programs defining agentic scaffolds, using them to solve math and science problems. Chan et al. (2024) test how well an LLM-based automated coding pipeline can perform in machine learning competitions.

Many program search systems search over code space by feeding programs into language models and asking them to evolve, crossover, and recombine them, thereby doing code evolution. However, these pipelines consist of many 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 (Novikov et al., 2025; Sharma, 2025; Lange et al., 2025). These design choices often are not ablated, nor are the code evolution systems compared to simple baselines, resulting in methods that are not methodically built up and are hence potentially suboptimal.

To systematically test from the ground up what matters in code evolution, we propose two simple baselines and compare them to several systems over three different domains.[1] Each domain tests how methods perform under a different constraint, such as a limited API budget when finding mathematical bounds, number of function evaluations when designing agentic scaffolds, or wall-clock time in machine learning competitions. Figures 1b and 1c illustrate the baselines, with the simplest being a form of random search, sampling IID from a language model while prompting it to solve a given problem. The second baseline is designed to better handle sequential problems. This

[*]Work partially done during an internship at Sakana AI. Email: yg@robots.ox.ac.uk
[1]Although the baselines are also a form of code evolution, we use the term "code evolution" to refer to more complicated methods unless specified otherwise.

baseline extends the first by conditioning on some problems generated in a previous generation, and optionally after a set number of generations, restarts from scratch.



(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 IID sampling a set of programs 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 those that ran successfully. Number of generated programs is only for the setup when searching over mathematical bounds.

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, finding problems in either how code evolution is implemented or used. When finding mathematical bounds, we find that both the domain knowledge put into the prompts and the expert-designed search spaces are more important for improving the bounds than the code evolution pipeline itself. We find that reformulating a problem's search space can improve bounds much more than code evolution does by itself, with different search pipelines finding similar bounds for a given search space. When evolving agentic scaffolds, we find that code evolution tends to select suboptimal scaffolds due to the datasets being relatively small and hence having a high variance. To keep API costs low, scaffolds are typically designed while evaluating ~ 100 examples (Hu et al., 2024), but this results in all generated scaffolds generalizing worse than a simple majority vote.

Our contributions are as follows:

- We introduce two simple baselines that perform well on a variety of domains, matching or exceeding purpose-built code evolution algorithms when given the same API budget, number of function evaluations, or wall-clock time.
- We show that when using code evolution to find mathematical bounds formulating a problem differently, thereby changing its search space, determines a pipeline's performance ceiling, while variable domain knowledge in the prompts can change its efficiency.
- When using code evolution to automatically design agentic scaffolds, we find that selected scaffolds tend to be overfit due to using small validation sets. We introduce a set of best practices to robustly evaluate scaffolds while keeping the search economical.
- We conclude with a discussion of best practices and open problems for future work. All code is released to facilitate baseline comparisons and the recommended practices.

2 Background

We briefly present code evolution here, with additional related work discussed in Appendix A. Code evolution pipelines typically work by starting with some program and using an LLM to generate new versions thereof. Often previous programs are given in the LLM's prompt, selected to maximize diversity or fitness. This is broadly how systems like AlphaEvolve (Novikov et al., 2025), ShinkaEvolve (Lange et al., 2025), OpenEvolve (Sharma, 2025), and ADAS (Hu et al., 2024) work. Some of these systems have many components, often requiring thousands of lines of code.

One setting where code evolution is used is in finding bounds for the circle packing problem, where given n circles one must maximize the sum of their radii while ensuring the circles do not overlap and all fit in a unit box. 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. Figure 1a illustrates this process. 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 for them and returning the best solution found or by constructing them directly.

Previous works using code evolution for scientific 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. (2025) discuss how occasionally injecting human priors into their code helped them find better matrix multiplication algorithms. Georgiev et al. (2025) 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 it finding an improved bound or 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 then with some domain-specific tweaks, e.g. Greenblatt (2024). 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

Both baselines are illustrated in Figure 1 *(b,c)*. The first baseline, IID random sampling (**IID RS**), consists of prompting a language model to produce code that solves some task. After running all programs, the best one is picked, either based on the bound it found for mathematical discovery or on some validation performance for agentic scaffolding and machine learning competitions.

The second baseline is a modification of the first which is designed to better deal with problems requiring iterative improvements. For example, in the kissing problem, one of the problems tackled in Novikov et al. (2025), one must find the largest possible set of vectors fulfilling some property. Often LLMs construct the solution directly by specifying these vectors and, at least in their initial answer, do not try extending known constructions. Thus, on similar problems, iterative sequential methods would likely perform better.

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 those 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 being less parallel than IID RS. The loss of parallelism is significant in cases where the main bottleneck is wall-clock time and not API cost, function evaluations, or compute.

3.1 How should methods be fairly compared?

When comparing methods, different ones 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 different solutions can differ due to numerical precision or 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.

For fairness, different methods should use a similar amount of domain knowledge in their prompts. Previous knowledge can be integrated by giving hints through the prompts, e.g. by describing properties of good or bad solutions. To keep comparisons fair, we use similar prompts across all methods unless mentioned otherwise, either those used in original works or new ones that are designed to have minimal domain knowledge beyond a problem's specifications. Some methods require different prompts due to how their pipelines are designed, e.g. when using the IID RS baseline we remove references to iteratively improving a solution.

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. (2025). 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 2 in Appendix B, with longer explanations in Novikov et al. (2025).

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. 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 $\geq M$ means 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
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>
# problems $\geq$ ShinkaEvolve		7/9		4/9	6/9
# problems $\geq$ AlphaEvolve			4/9	2/9	4/9
Average rank			1.89	2.28	1.83

As AlphaEvolve is closed-source, we compare to a similar open-source sample-efficient pipeline, ShinkaEvolve (Lange et al., 2025).[2] For a fair comparison, the baselines and ShinkaEvolve use the same minimal-domain-knowledge prompts. ShinkaEvolve is restricted to Gemini-2.5 Pro, Flash, and Flash Lite, akin to AlphaEvolve, while both baselines use Gemini-2.5 Pro, so differences in performance are not due to different model families. Additional technical details are in Appendix C.

Results are reported for all methods using a 20\$ budget. Under these conditions ShinkaEvolve runs for 500-800 generations, or $\sim$50 generations per dollar. For reference, the circle packing run in Lange et al. (2025) costs about \$12 and ran for 150 generations (12.5 generations per dollar). The baselines were run beyond this budget to allow estimating uncertainties. Comparisons are then done by taking the first chronologically generated results within the \$20 budget or sampling equal-budget subsets when calculating uncertainties, see Appendix F for details. Only a single ShinkaEvolve run is used per problem as these experiments are very expensive – running ShinkaEvolve and the two baselines costs >\$70 per problem. The baselines are easier to slightly oversample as they come in smaller discrete units than a full Shinka run, being more regular samples for IID RS and more trials for SCS. Oversampling the baselines results in most runs costing \$25 to \$30, with the most expensive problem costing almost \$50 per run.

Results. 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.

[2]We do not compare to OpenEvolve due to difficulties getting it to run under the same conditions. A partial comparison is in Appendix D.

[3]Using everything the baselines generated, 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.

Including domain knowledge in a prompt can help boost a method's performance without changing its pipeline. For example, to make a fair comparison, ShinkaEvolve is initiated from a minimal, uninformative initial program. For circle packing, ShinkaEvolve and OpenEvolve's original initial program includes a function that finds the maximum circle radii given their centers. Using a prompt that includes a similar function allowed IID RS to find a circle packing with the same score of 2.63598 when sampling 1000 programs. The two prompts are compared in Appendix E.

On the other hand, domain knowledge can also prove detrimental if it biases a search method down a bad trajectory. We show this by comparing ShinkaEvolve's circle packing performance when using a minimal initial program, consisting of a fixed placement of tiny circles, to the original one used by Lange et al. (2025); Sharma (2025), where the initial program places circles in a set of rings and finds their max possible radii. When initialized from the minimal program, ShinkaEvolve found the same bound of 2.63598 over three runs, whereas two out of three runs using the original program found subpar circle packings. Using the original program in conjunction with ShinkaEvolve's original prompt, which specifies characteristics of typical good and bad solutions, enables it to find the same bound for three out of three runs.

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 F. Figure 2 demonstrates that both baselines compare favourably 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. This and other mechanisms, such as occasionally using cheaper LLMs, do not seem to make ShinkaEvolve significantly more cost-efficient.

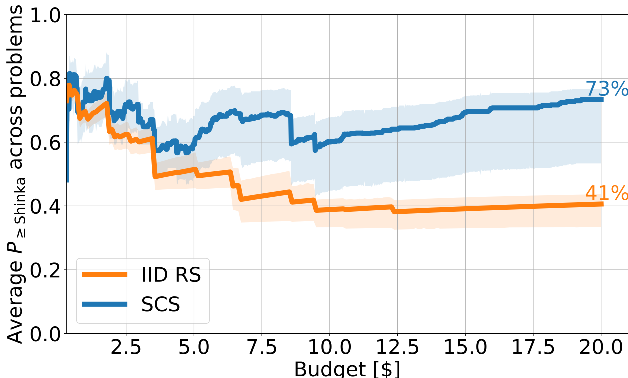


Figure 2: Average probability of matching or exceeding ShinkaEvolve over the 9 problems for our two baselines. Numbers on the right are the probabilities at the max budget of $20. Per problem breakdown is in Appendix G. Both baselines perform well over different budgets, with sequential conditioned sampling (SCS) generally outperforming ShinkaEvolve. Shaded regions are asymmetric 95% confidence intervals, see Appendix F for details.

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 $\sim$10 hours. In Appendix H we find that the baselines are sample efficient also with respect to the number of function evaluations.

4.1 A PROBLEM'S SEARCH SPACE DICTATES ITS PERFORMANCE CEILING

As shown in Table 1, for some problems, all methods find a similar bound. Therefore, given a sufficient budget, all methods can likely reach the same performance ceiling. It is unclear whether the discovered bounds are thus tight, or whether the search pipelines all converge at similar suboptimal solutions.

To test this, we take one of the problems and improve its formulation. Changing a problem's formulation equates to 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 I. Note that AlphaEvolve's formulation is from Gonçalves et al. (2017), with AlphaEvolve 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 optimizing a given setup, regardless of the search pipeline.

Notably, the 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 sophisticated search pipelines are arguably redundant. The baselines

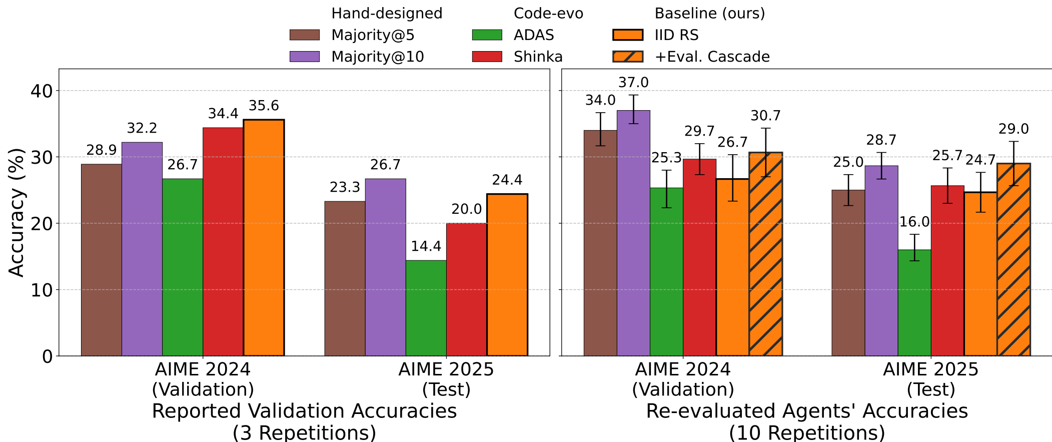


Figure 3: AIME 2024 and 2025 accuracies for different methods. 2024 was used as the validation set, with 2025 serving as a test set. Majority@5/10 indicate manually designed majority vote scaffolds. **(Left)** Validation accuracies are those measured while evolving different scaffolds, with both validation and test accuracies being for 3 evaluations over the dataset. ShinkaEvolve numbers are from Lange et al. (2025). 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 moreso due to stochasticity in the evaluations than them achieving good performance. Unlike when evaluating only 3 times, here 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 better scaffold and one that generalizes more, being essentially equal to majority vote@10 on the test set (49.5% probability of improvement).

achieving comparable performance to ShinkaEvolve, and on some problems to AlphaEvolve, further supports this.

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, 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 (Hu et al., 2024; Lange et al., 2025). Typically, the limiting factor is not meta-agent queries but how many scaffolds can be evaluated as each evaluation requires many LLM calls, typically costing $1-10 per evaluation depending on which LLM is used. For example, Hu et al. (2024) mentions 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 (Hu et al., 2024) 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, while 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 on designing scaffolds for questions from the AIME math competition, with all agents using GPT4.1-nano for their scaffold evaluations. 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 J.

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 the validation sets used to find scaffolds are typically constructed of $\sim$100 questions, including repetitions, regardless of the full dataset's size (see Hu et al., 2024, Appendix E). This is to keep the already costly evaluation economical. However, it introduces 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.

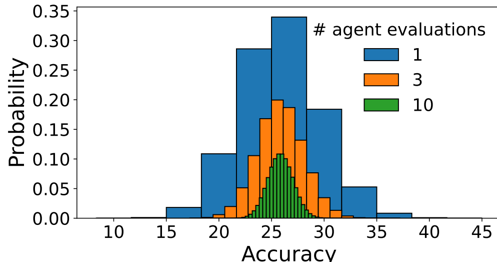


Figure 4: Empirical distributions of majority vote@5 accuracies for AIME 2025 when evaluating each question a different number of times. Even when evaluating 10 times there is a standard deviation of more than 1% in the accuracy. Distributions were calculated by sampling 100 answers to each question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

Stochasticity Reduction. Reducing this 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 approximately > 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, methods should 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.

When comparing methods' test accuracies, we recommend using the probability of improvement. Introduced by Agarwal et al. (2021) 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. See Appendix K for an extended discussion.

These recommendations are only for final evaluations – how should scaffolds be evolved? Using better comparison metrics such as the probability of improvement and larger effective validation sets would not wholly fix the problem. The bias towards accidentally picking worse scaffolds due to the evaluation's stochasticity would still exist, while simply increasing the number of evaluations is expensive. For cost reduction specifically, it is more economical to use an evaluation cascade. In a two-level cascade, bad scaffolds can be evaluated over a small effective validation set while those that are potentially competitive are re-evaluated over a larger set to see if their improvement holds. To 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, instead of a single other method. See Appendix K for a definition and discussions.

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. However, while this scaffold outperforms scaffolds found without an evaluation cascade, it does not outperform majority vote@10.

6 MACHINE LEARNING COMPETITIONS

Technical details and the setup for the MLE bench experiments are deferred to Appendix L. There, Table 4 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 also perform well in time limited domains.

7 DISCUSSION

This paper demonstrates that across several domains simple baselines compare to, if not exceed, domain-specific sophisticated code evolution pipelines. Meanwhile, each domain has a different constraint – a limited API budget when finding mathematical bounds, a limit on the number of function evaluations when designing agentic scaffolds, and limited wall-clock time in machine learning competitions. Across domains we find various insights and problems, some contrary to common wisdom. Simple baselines performing well shows that code evolution systems need not be complicated to get good results, with other factors mattering more. For finding mathematical bounds, the domain knowledge used and a problem's formulation affect the search's efficiency and performance ceiling much more than the pipeline. When designing agentic scaffolds evaluations are typically over small datasets, leading to high stochasticity and the resulting scaffolds being suboptimal. This results in simple manually crafted scaffolds, like a majority vote agent, outperforming automated search systems.

Why were these shortcomings not identified previously? This is discussed in detail in Appendix O, but briefly, they seem to stem from code evolution's nascent state as a field. Methods and benchmarks are continuously developed. Rigorous best practices lagging behind leads to results not being as strong as they may initially seem.

The importance of a problem's defined search space is interesting with respect to some claimed goals of code evolution. The simple baselines performing well in section 4 imply that the problems there might be inherently relatively easy to search over, with section 4.1 showing that more significant improvements can come from improving a problem's verifier. A different verifier implicitly defines a different search space, as it changes the function being optimized but not the underlying task of finding an improved bound. Hu et al. (2024) 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? While it is technically possible to reinvent advanced mathematics and check for a different formulation's correctness within a computer program, this has not been observed in practice. It would be interesting in future work to design systems that are flexible enough to change their problem formulations or find new ones.

Many improvements introduced in code evolution pipelines are potentially useful in one setting but not another, so universally keeping them could be detrimental. While textual feedback is important for tasks like MLE bench, it might increase costs without resulting in substantial performance improvements in other settings. In this sense, it is unclear whether there is as of yet a code evolution agent that can perform well universally, beyond the capabilities trained into its base model.

In conclusion, future works should:

- Ensure methods are fairly compared, using the same language models, domain knowledge in prompts, verifiers and hence search spaces, and budgets. It is important to be specific about a problem's main constraint, whether it is an API cost, function evaluations, wall-clock time, or something else entirely.
- When designing agentic scaffolds, make sure to evaluate a sufficient number of samples during both evolution and test time, compare methods using a probability of improvement or the probability of dominance, and report 95% confidence intervals to ensure results are not spurious. While searching for scaffolds, use an evaluation cascade to keep the search economically feasible while minimizing stochasticity.
- Be clear whether the work proposes a search method or solely demonstrates a scientific discovery. For scientific discovery, the main research contribution may be not the search pipeline but the domain knowledge used, both in the prompt and in designing the search space. When proposing search methods, use simple baselines!

IMPACT STATEMENT

Better baselines and experimental methodology can help improve scientific rigor and produce advancements in code evolution. However, this can also lead to a false pretense of results being better if comparisons are not done properly, see Figure 3. Better statistical testing should be done thoughtfully as otherwise it is prone to common misinterpretations and various pitfalls, such as p-hacking.

Following Agarwal et al. (2021), we recommend avoiding tests that use p-values or have binary significant/non-significant outcomes.

While improvements in code evolution can result in new scientific discoveries, these improvements come with various unknown and potentially negative effects on society. Automation can lead to job loss and other social problems that would need to be dealt with by policymakers.

Code evolution research can be expensive, especially when requiring more baselines and thorough evaluations. This could make it difficult for some communities to partake in this work. To mitigate this, in Appendix N we discuss ways to make code evolution development cheaper, so it can be more accessible while still being scientifically rigorous.

REFERENCES

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.

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.

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.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

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

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

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.

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.

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

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.

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

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

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.

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, pp. 1077–1082. Morgan Kaufmann Orlando, 1999.

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

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.

Chris Lu, Samuel Holt, Claudio Fanconi, Alex Chan, Jakob Foerster, Mihaela van der Schaar, and Robert Lange. Discovering preference optimization algorithms with and for large language models. *Advances in Neural Information Processing Systems*, 37:86528–86573, 2024.

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.

Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C Landsness, Daniel L Barabasi, Siddharth Narayanan, Nicky Evans, et al. Kosmos: An ai scientist for autonomous discovery. *arXiv preprint arXiv:2511.02824*, 2025.

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.

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.

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.

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

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.

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.

A RELATED WORK

Although modern code evolution relies on LLMs, code evolution as a term has existed at least as far back as 1999 (Keller & Banzhaf, 1999). Although only an emerging field, there are a plethora of works on code evolution, with this work focusing on a representative few (see Appendix P for an extended discussion). Some earlier LLM-based code evolution methods have components similar to the two given baselines. FunSearch (Romera-Paredes et al., 2024) and AlphaCode (Li et al., 2022) 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. (2023) use an SCS-like code evolution pipeline to find better reward functions for reinforcement learning environments.

Various existing works discuss sample-efficient code evolution. Lange et al. (2025) present ShinkaEvolve, a code evolution pipeline that is claimed to be sample-efficient and open-ended. Our experiments show that its sample-efficiency might stem not from the pipeline but auxiliary factors, such as its prompts. Chen et al. (2023) show that in some classic programming settings teaching a language model to debug significantly improves its sample efficiency. This is in-line with our observations in section 6, where without debugging code evolution pipelines rarely produce a valid program.

In other subfields, especially reinforcement learning, there have been many efforts to improve the field's robustness. Henderson et al. (2018) discuss difficulties in reproducing existing work in RL while Agarwal et al. (2021) showcase problems in evaluations, proposing various best practices. Goldie et al. (2025) compare a variety of meta-RL algorithms, discussing best practices for future work.

Regarding agentic scaffolds, El et al. (2025) 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 2 lists the 9 problems used in section 4, taken from Novikov et al. (2025).

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 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 append 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 (Sharma, 2025) 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

Table 2: 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. (2025). "Pre-AE" are the best bounds from before Novikov et al. (2025).

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

budget for a run. Table 3 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, compared to the baselines and ShinkaEvolve OpenEvolve got an average rank of 3.08.

Table 3: 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 DIFFERENT CIRCLE PACKING PROMPTS

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.
```

The prompt used for the IID RS domain knowledge experiment mentioned in section 4 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]×[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
from typing import Tuple
from scipy.optimize import linprog

def verify_circles(circles: np.ndarray) -> bool:
    """Checks that the circles are disjoint and lie inside a unit square."""
    for circle1, circle2 in itertools.combinations(circles, 2):
        center_distance = np.sqrt((circle1[0] - circle2[0])**2 + (circle1[1] -
        ↪ circle2[1])**2)
        if center_distance < circle1[2] + circle2[2]:
            return False
    for x, y, r in circles:
        if x - r < 0 or y - r < 0 or x + r > 1 or y + r > 1:
            return False
    return True
```
```

```
def compute_max_radii(centers):
    n = len(centers)
    centers = np.array(centers)

    # upper bounds from boundary constraints
    u = np.min(np.vstack([centers[:, 0], 1 - centers[:, 0],
                          centers[:, 1], 1 - centers[:, 1]]), axis=0)

    # Objective: maximize sum r_i -> minimize -sum r_i
    c = -np.ones(n)

    # Constraints A_ub @ r <= b_ub
    A = []
    b = []

    # boundary constraints: r_i <= u_i
    for i in range(n):
        row = np.zeros(n)
        row[i] = 1.0
        A.append(row)
        b.append(u[i])

    # pairwise non-overlap constraints: r_i + r_j <= d_ij
    for i in range(n):
        for j in range(i + 1, n):
            dij = np.linalg.norm(centers[i] - centers[j])
            if dij < u[i] + u[j]:  # only add if potentially active
                row = np.zeros(n)
                row[i] = 1.0
                row[j] = 1.0
                A.append(row)
                b.append(dij)

    A = np.array(A)
    b = np.array(b)

    # bounds: r_i >= 0
    bounds = [(0, None) for _ in range(n)]

    res = linprog(c, A_ub=A, b_ub=b, bounds=bounds, method="highs")

    if not res.success:
        raise RuntimeError("LP solver failed: " + res.message)

    radii = res.x
    return radii
```
```

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 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. (2021). 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 5, is 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.

G FIGURE 2 PER-PROBLEM BREAKDOWN

See Figure 5.

H MATH BOUNDS PER-PROGRAM EVALUATION SAMPLE EFFICIENCY

Figures 6 and 7 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 6 being lower than those in Figure 2, but only slightly.

I IMPROVED UNCERTAINTY INEQUALITY FORMULATION

We first describe the problem in its generality, based on Appendix B.4 of Novikov et al. (2025) and (Gonçalves et al., 2017, 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 nonegative be defined as $A(f) := \inf(\{r > 0 | \forall |x| \geq r : f(x) \geq 0\})$. In the uncertainty inequality problem we wish to find the smallest 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]

Denoting the nth Hermite polynomial as H_n, Gonçalves et al. (2017, 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 α_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).

Gonçalves et al. (2017) construct their lower bound of 0.3523 by setting all α_n except for $\alpha_0, \alpha_1, \alpha_2, \alpha_3$ to zero and numerically finding which αs minimize C. This is the formulation also used by Novikov et al. (2025) and in our Table 1.

[4]For this last condition Novikov et al. (2025) mistakenly state it as < 0, see Gonçalves et al. (2017, p. 679).

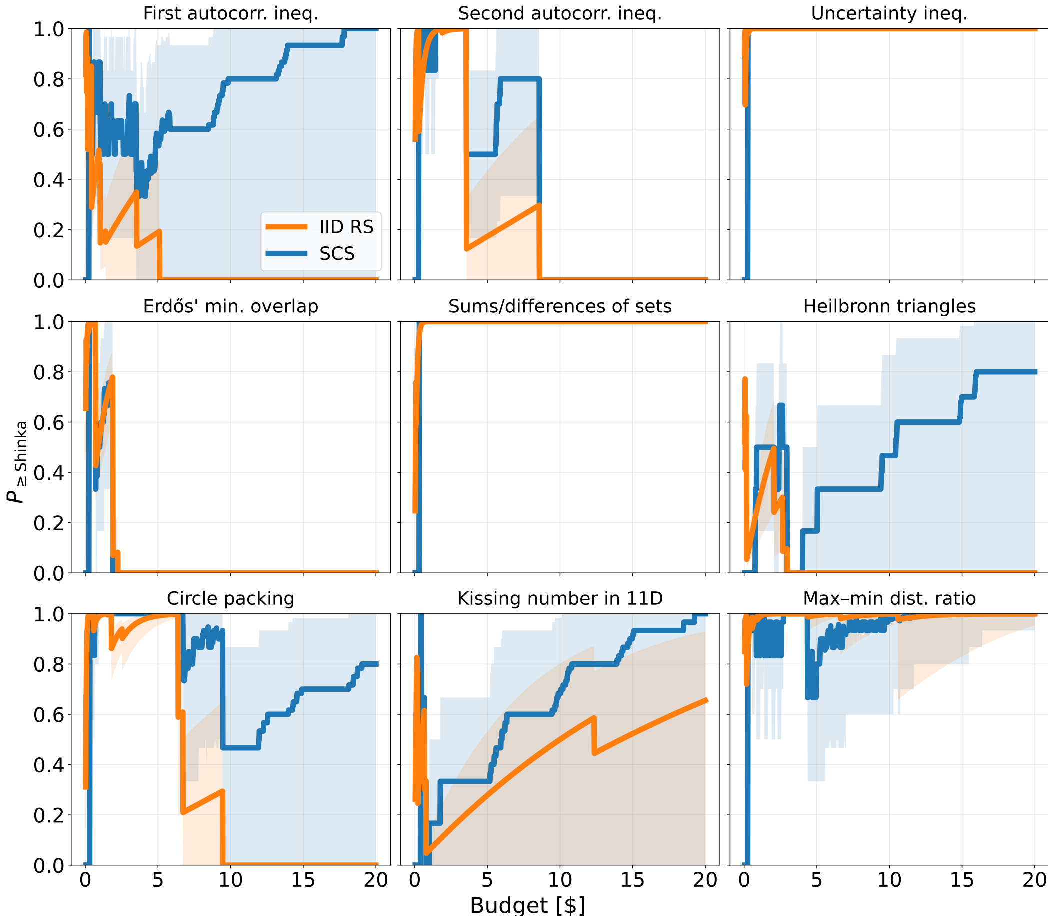


Figure 5: Per-problem probability of matching/exceeding ShinkaEvolve for the two baselines. Shaded regions are 95% confidence intervals.

We modify this formulation in two ways. First, Novikov et al. (2025) 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 αs beyond α_7 to zero instead of α_3. This allows $f(x)$ to represent a larger class of functions. It is likely possible going beyond α_7 and reducing the bound further but we encountered numerical instabilities when trying to do so, likely from using very high order polynomials.

J AGENT EVOLUTION TECHNICAL DETAILS

Lange et al. (2025) 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.

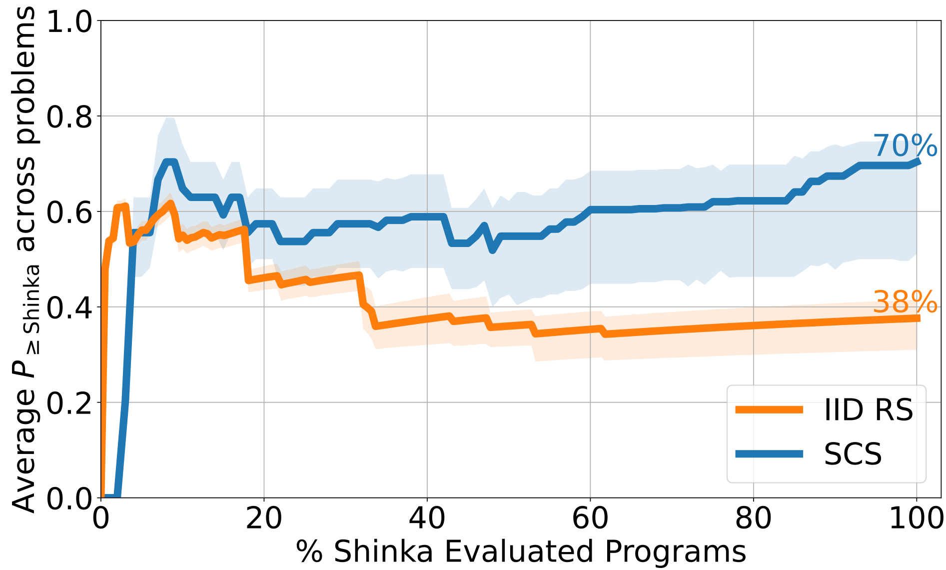


Figure 6: For each baseline, the average probability of matching or exceeding ShinkaEvolve across the 9 problems, as a function of the number of evaluated programs. The x axis is the percent of evaluated programs out of the maximum in the corresponding ShinkaEvolve run. Shaded regions are 95% confidence intervals.

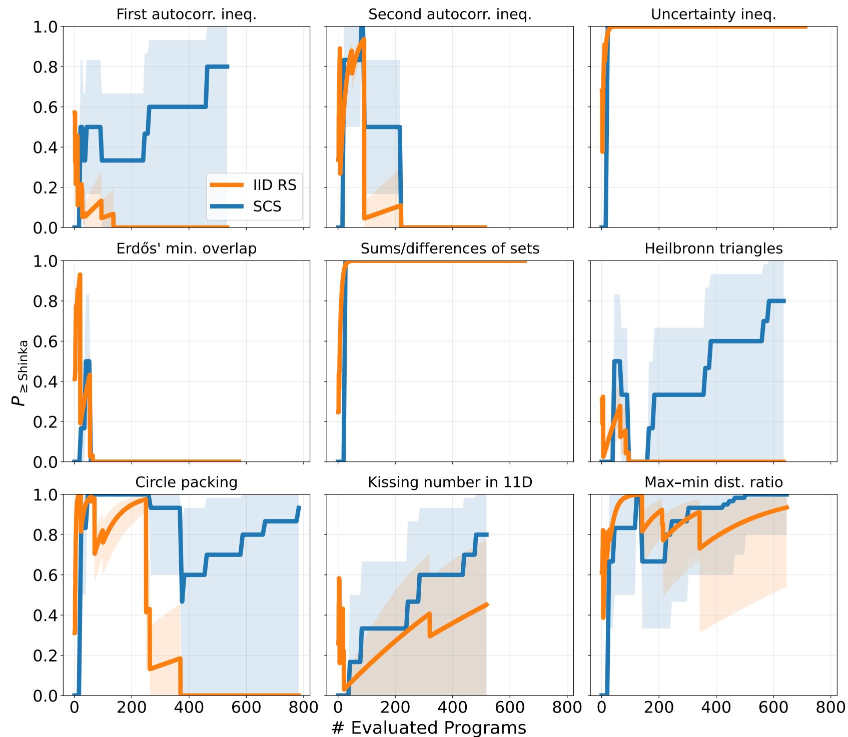


Figure 7: For each baseline, the per-problem probability of matching or exceeding ShinkaEvolve across the 9 problems, as a function of the number of evaluated programs. Shaded regions are 95% confidence intervals.

K 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. (2021) 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. (2021) 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

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

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 point estimates. 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.

L MLE BENCH RESULTS AND TECHNICAL DETAILS

To see how the baselines compare to code evolution for tasks with a limited wall-clock time we test their performance on MLE bench, a benchmark of Kaggle competitions (Chan et al., 2024). 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 reported runs are given access to a single RTX 8000 GPU with 12 CPU cores. We use the ten-competition subset of Schmidgall et al. (2025) for evaluating the different methods.

We compare the baselines to AIDE, a code evolution pipeline designed specifically for Kaggle competitions (Jiang et al., 2025). Chan et al. (2024) find that on MLE bench AIDE outperforms other code evolution systems, such as MLAB (Huang et al., 2023) and OpenHands (Wang et al., 2024).

L.1 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. (2025).

Table 4: Human baseline metrics and results for different methods on the Kaggle competitions. SCS matches or exceeds AIDE on 6/10 competitions while IID RS matches or exceeds it on 3/10. Median is the median score in the competition, with bronze, silver, and gold denoting the cutoff score to receive each medal. $*, +, \times$ 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. Human baseline metrics are from Schmidgall et al. (2025). Unabbreviated competition names are given in Appendix M.

	Human baseline metrics						
Competition	Median	Bronze*	Silver^{+}	Gold$^{\times}$	AIDE	IID RS	SCS
Insult Detection (↑)	0.778	0.791	0.823	0.833	**0.9135**$^{\times}$	0.9056$^{\times}$	0.8460$^{\times}$
Dec. 2021 Tab. (↑)	0.953	0.956	0.956	0.956	0.9620$^{\times}$	0.9621$^{\times}$	**0.9623**$^{\times}$
Trans. Conduct. (↓)	0.069	0.065	0.062	0.06	0.0619^{+}	0.0620^{+}	**0.0616**$^{+}$
Engl. Txt (↑)	0.990	0.990	0.991	0.997	0.9823	0.9332	**0.9913**$^{+}$
May 2022 Tab. (↑)	0.972	0.998	0.998	0.998	**0.9965**	0.9140	0.9839
Random Pizza (↑)	0.599	0.692	0.724	0.979	0.6820	0.6206	**0.6891**
Spooky Author (↓)	0.418	0.293	0.269	0.165	**0.2883***	0.4468	0.4031
Toxic Jigsaw (↑)	0.980	0.986	0.986	0.987	0.9807	0.9780	**0.9863**$^{+}$
Russ. Txt (↑)	0.975	0.975	0.982	0.990	**0.9756***	0.9735	0.9453
NYC Taxi (↓)	3.597	2.923	2.881	2.337	11.460	**5.1483**	5.9758
# competitions $\geq$ Median					8	4	8
# competitions $\geq$ Bronze					5	3	5
# competitions $\geq$ Silver					3	3	5
# competitions $\geq$ Gold					2	2	2

Notably, due to the competitions' intricacy, zero-shot generated programs almost never run without errors. This is due to there being many 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. Unlike AIDE, the baselines do not include explicit fitness-based selection or a memory of all past programs and their performance.

M FULL MLE BENCH COMPETITION NAMES

Table 5 lists the unabbreviated names of the competitions in Table 4.

Table 5: Competition abbreviations and full names.

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

N Tips for economically feasible 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 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. (2025) use a Deepseek model instead of API calls, although they do it 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 still require renting GPUs if there are none available.

Second, when evaluations are expensive, there are ways to make them cheaper. The evaluation cascade described in section 5 helps keep the cost of agent evaluations low while still reducing stochasticity. Schmidgall et al. (2025) 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.

O Why were the discovered shortcomings not seen previously?

First, in many code evolution works, different contributions are conflated, so baselines are often ignored. For example, Novikov et al. (2025) give two main contributions: they both introduce a search method, AlphaEvolve, and also demonstrate various scientific discoveries and optimizations it helped them make. Both are likely valuable in and of themselves, but a good scientific discovery does not necessarily mean that the method used to find it is performant. Thus, it is important to be clear when proposing search methods and when exhibiting a discovery. Search methods should be benchmarked under fair conditions and relative to simple baselines. Scientific discoveries should be clear about what expert knowledge was required for the search pipeline to achieve its result.

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. Lange et al. (2025) compares to several other methods' published results and sample efficiencies on circle packing, but do so while using different prompts and hence domain knowledge. This is largely due to the nascent state of code evolution as an emerging field, with differences between prompts potentially being seen as unimportant. In general, methods should be compared using as similar of a setup as possible. The baselines in our work provide an exciting opportunity from which to methodically build principled methods, where each component is ablated and has a clear contribution. There are other kinds of accidental unfair comparisons we did not demonstrate but likely exist. For example, on search problems – like finding mathematical bounds – tuning hyperparameters can artificially improve a method's sample efficiency, as the budget used for the tuning should count towards the overall search.

Another reason why some of these shortcomings were missed is that code evolution is expensive and time-consuming to run, necessitating the development of better benchmarks. Some benchmarks, like MLE bench, are very expensive to fully evaluate, with Toledo et al. (2025) using an estimated 50k-100k H100 hours to thoroughly compare several different methods. This limitation applies to many works using code evolution, including this one, with it and other limitations discussed in Appendix P.

P Limitations

Although this work aims to enable better comparisons in code evolution, its main limitation is its lack thereof, this being due to a few reasons. First, many code evolution pipelines are designed and demonstrated over specific purposes, such as finding better rewards in RL environments (Ma et al., 2023), scientific discovery (Mitchener et al., 2025), and discovering preference optimization algorithms (Lu et al., 2024), with some of these pipelines not being open-source. As there is relatively little standardization, setting up and comparing a method on a new setup requires nontrivial effort. Moreover, drops in performance are then not clearly from a search method underperforming but

potentially due to misapplying it. Greater standardization in code evolution, across benchmarks and tasks, would enable better comparisons in future work.

In addition to this, code evolution suffers from being costly to run, thereby limiting the number of experiments which are economically feasible. We discuss ways to make it cheaper in Appendix N, but these are not yet standard within the community. Evaluation costs are why most experiments chiefly use Gemini models as well. Shifting to cheaper evaluation and benchmarking methods would additionally allow better comparisons in future works.

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 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 found bound 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 well known.