LEARNING TO DISCOVER EQUATIONS BY EVOLVING SEARCH ALGORITHMS

Anonymous authors
Paper under double-blind review

ABSTRACT

Symbolic regression (SR), the problem of discovering interpretable equations that explain observed data, is traditionally done with evolutionary search using hand-designed heuristics. We introduce a meta-evolutionary method that uses LLMs to discover and improve symbolic regression algorithms. Our method addresses stochastic evaluation of search algorithms with a reevaluation strategy integrated into the evolutionary loop, reducing selection based on chance successes. We apply meta-evolution to evolve symbolic regression algorithms from a minimal genetic algorithm skeleton and improve an existing popular, high-performance SR algorithm, PySR. The resulting algorithms are reusable across new problems and require no LLM calls at inference time. The evolved PySR algorithm improves ground truth equation recovery from 50.6% to 58.6% on SRBench, and improvements transfer to naturalistic equation discovery tasks (EmpiricalBench) and to a mechanistic interpretability application. We also show how meta-evolution can be used to fine-tune algorithms for specialized domains, showing increased performance for neuroscientific and mechanistic interpretability equation recovery, even with limited training data.

1 INTRODUCTION

Discovering interpretable equations from raw data has been a goal for artificial intelligence from its early days (Langley, 1981) to the present (Murphy, 2026), and much earlier within the empirical sciences (Kepler, 1609). Equation discovery, particularly *symbolic regression*, holds practical value for the natural and social sciences, while also engaging with a core challenge in modern AI—making accurate predictions *and* rendering interpretable explanatory value. A typical symbolic regression problem comprises a low-dimensional tabular dataset paired with a set of algebraic operators, and its solution consists of equations which explain the data using those operators. This gives practitioners flexibility to inject prior knowledge in the form of loss functions (Dotson et al., 2025), constraints on how those operators can combine, or the inclusion of custom operators tailored to the domain.

However, the equation search space is combinatorial and exponential, making any exact method infeasible. The field has coalesced around heuristic discrete search, such as genetic algorithms, leading to a proliferation of ad-hoc search moves, novelty bonuses, fitness and selection mechanisms, etc. (Poli et al., 2008; Lehman & Stanley, 2011; Cranmer, 2023), none of which universally dominates (de Franca et al., 2025). Learning to search might seem like a natural answer: training neural nets to generate equations combined with tree search (Li et al., 2019; Kamienny et al., 2022). But training a custom neural network loses the flexibility to work in *new* search spaces unseen during training, with different dimensionalities, data ranges, algebraic operators and constraints, which is how symbolic regression is used in practice. Inserting a neural network into the inner loop of search also introduces a tradeoff between the model's computational cost and its accuracy: A small model may beat a large one, simply because the smaller model can churn through more candidate equations, even if the small model is less accurate.

Here we introduce a new method for learning to search for equations, based on program synthesis. The idea is to search for a program which searches for equations. An expensive neural network manages the outer loop of search (exploring different programs), while the inner loop of search remains cheap because it executes fast high-level source code (exploring different equations). After learning, no new neural network calls are needed, amortizing the cost of program synthesis. The end

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

result is a high-level program inheriting the full generality of classic symbolic regression: It works with any set of algebraic operators, tabular data of any dimensionality or numerical range, and can be adapted to custom loss functions, all of which are prerequisites to real-world use.

Our method (Figure 1) works by optimizing a program against a training set of symbolic regression problems. This program is reused on new problems, therefore amortizing the cost of optimization. This however raises the question of whether the learned program generalizes to qualitatively different search problems, and whether its learned heuristic search can extrapolate to the longer runtimes needed to solve hard problems. We answer these questions in the affirmative, though the picture is nuanced: Ultimately these discrete search problems are typically at least NP-hard so we should not expect a universal heuristic solution. Nonetheless, we find our method transfers well and the evolved PySR retains its advantage under longer search budgets.

Our method can be seen as LLM-guided program synthesis in the spirit of AlphaEvolve (Novikov et al., 2025) and others (Romera-Paredes et al., 2024; Sharma, 2025; Agrawal et al., 2026; Cemri et al., 2026). However, our approach raises a core technical problem understudied in LLM-guided program synthesis, because our objective function is stochastic: We seek a stochastic search program which, on average, quickly finds good equations. We analyze the effect of this stochasticity, finding that evolution without reevaluation suffers, while reevaluation improves the quality of the discovered programs. In total we contribute the following:

1. A method for learning to discover equations based on an LLM-guided program synthesis algorithm that handles stochastic quantitative objectives.
2. A study of the method on problems from several fields— physics, neuroscience, and mechanistic interpretability— finding our method transfers across domains, and that it can be 'pretrained' on one domain and 'fine-tuned' on another, in case data are sparse.
3. A study of how the method transfers across different symbolic regression objectives, such as training to discover exactly the ground-truth equation, but testing on mean-squared error of the discovered equation. Counterintuitively, we find less transfer across objectives than across domains.
4. An improved version of PySR (Cranmer, 2023) automatically created by our method. PySR is the most used symbolic regression algorithm, and we are working on integrating our discovered results into its codebase, making it accessible to the many thousands of scientists who use PySR.

2 Related work

Symbolic regression. Symbolic regression (SR) approaches traditionally use evolutionary algorithms, but approaches based on deep learning (Kamienny et al., 2022; Biggio et al., 2021; Holt et al., 2023; Petersen et al., 2021) and hybrid approaches combining neural networks and search have also been explored (Mundhenk et al., 2021; Yu et al., 2025; Kamienny et al., 2023). While neural guidance can reduce search time, generalization outside of the training distribution remains a challenge as SR problems cover a wide space of operators, functions, and input domains. Evolutionary SR algorithms remain highly competitive and widely used (La Cava et al., 2021; Cranmer, 2023). More recently, researchers have proposed SR methods integrating large language models (LLMs) into the equation search (Grayeli et al., 2024; Shojaee et al., 2025), improving performance but increasing cost per task due to the LLM queries taken in the search loop. Our work instead integrates LLMs in an outer loop of search during training to discover evolutionary SR algorithms that can be run at test time with no additional neural network queries.

LLM-guided program evolution. LLM-guided program evolution queries large language models in an evolutionary loop to optimize programs to achieve an objective (Romera-Paredes et al., 2024; Novikov et al., 2025). Follow-up work has proposed incorporating adaptive evolutionary strategies (Cemri et al., 2026; Liu et al., 2026) or test-time training (Yuksekgonul et al., 2026). Our work applies LLM-evolution to the problem of discovering SR algorithms. Evolving a search algorithm brings up the problem of stochastic and expensive evaluation. Noise in fitness evaluations has been studied for evolutionary algorithms in the past (Jin & Branke, 2005). We study reevaluation strategies for LLM-guided evolution of search algorithms, focusing on improving selection under a limited evaluation budget.

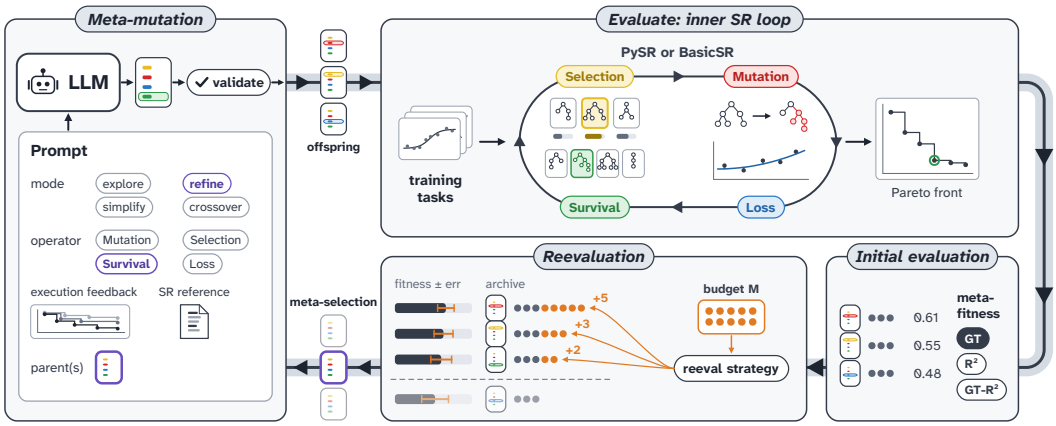


Figure 1: **Meta-evolution of symbolic regression (SR) algorithms.** An LLM rewrites one operator of a parent SR program, guided by a mutation mode, execution feedback, and a codebase reference. Each offspring runs as an inner SR loop on training tasks $\times$ seeds and is scored by a meta-fitness metric (ground-truth equation discovery 'GT,' curve-fitting R^2, or both, GT-R^2). Scoring each SR program is stochastic, necessitating multiple random seeds (Initial Evaluation) and reevaluation across meta-evolution (Reevaluation).

Learning to search and meta-evolution. Learning to search is a broad research area. Our setting is related to work with neurally guided program synthesis methods (Balog et al., 2017; Ellis et al., 2021), teaching transformers to search (Gandhi et al., 2024), and synthesizing program optimizations (Bunel et al., 2017). More specifically, meta-evolutionary algorithms have previously been explored to improve evolutionary optimization algorithms (Oltean, 2005). EvoX (Liu et al., 2026) conducts LLM-guided program evolution and periodically improves the evolutionary outer loop search strategy by selecting and mutating previously successful strategies. LLaMEA (van Stein & Bäck, 2025) is closely related to our work, using LLM-guided evolution to generate and refine heuristic optimization algorithms. We focus on discovering reusable algorithms for symbolic regression, both from a minimal algorithm skeleton and by improving an established high-performance solver, PySR.

3 Problem Statement

A symbolic regression problem comprises a search space $\mathcal{E}$ of equations $e \in \mathcal{E}$, a dataset $\mathcal{D}$, and an objective function $f_{\mathcal{D}}$ which scores equations against $\mathcal{D}$. The search space $\mathcal{E}$ is typically constructed by defining a set of algebraic operators ($\times$, $+$, sin, exp, ...), and sometimes further constrained by dimensional analysis, grammatical rules, or other constraints. The dataset $\mathcal{D}$ is typically real-valued and low-dimensional. The objective function is commonly multivalued: For example, $f_{\mathcal{D}} : \mathcal{E} \to \mathbb{R}^2$ could measure both mean-squared error and equation size, to jointly optimize parsimony and accuracy, i.e. $f_{\mathcal{D}}(e) = \left\langle \textsc{Size}(e), \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} (y - e(x))^2 \right\rangle$ is a common multivalued objective.

A symbolic regression solution S is a Pareto frontier of equations which optimize $f_{\mathcal{D}}$. The optimal solution $S^\star$ would be exactly those expressions which are joint minimizers of the objective[1]

$$S^\star = \arg\min_{e\in\mathcal{E}} f_{\mathcal{D}}(e). \tag{1}$$

A symbolic regression algorithm π is a randomized algorithm which outputs a solution S given a symbolic regression problem and a compute budget B, formally $\pi(\mathcal{E}, \mathcal{D}, f, B) \in \Delta(2^{\mathcal{E}})$. We consider both wallclock budget and budgeting the number of expression evaluations during search. The

[1]We define the multivalued argmin as $y \in \arg\min_{x\in X} g(x)$ iff for all $x \in X$ either $g(x) = g(y)$ or there is an index i where $g(x)_i > g(y)_i$ — the Pareto frontier.

canonical symbolic regression algorithm is a genetic algorithm with many heuristics: Hand-designed mutation and crossover samplers that randomly edit equations (PySR, for example, samples an ensemble of 11 custom mutation operators); selection rules such as tournament selection or roulette selection; islands of individuals that evolve independently and occasionally migrate; heuristic bonuses for an equation being novel; and many other tricks (Poli et al., 2008).

4 METHOD

Our aim is to synthesize or learn a program π for quickly finding good solutions to symbolic regression problems. We do this by optimizing π against a training set of symbolic regression problems, with the ultimate goal of generalizing to unseen test problems. To do this, we assume train/test corpora, $\mathcal{C}_{\text{TRAIN}}$ and $\mathcal{C}_{\text{TEST}}$, each equipped with a compute budget B and solution score $\mathcal{L}$, where higher values indicate better solutions S. At train time we maximize the expected fitness $J(\pi, \mathcal{C}_{\text{TRAIN}})$,

$$J(\pi, \mathcal{C}_{\text{TRAIN}}) = \frac{1}{|\mathcal{C}_{\text{TRAIN}}|} \sum_{(\mathcal{E},\mathcal{D},f,B,\mathcal{L})\in\mathcal{C}_{\text{TRAIN}}} \mathop{\mathbb{E}}_{S\sim\pi(\mathcal{E},\mathcal{D},f,B)} [\mathcal{L}(S)] \,. \tag{2}$$

As different executions of π can yield different solutions, the expected solution score must be approximated via sampling, giving stochastic estimates of J. Note also that the solution score $\mathcal{L}$ differs from the inner-loop objective f, which is minimized, because $\mathcal{L}$ can utilize privileged information such as holdout data or ground-truth equations—hidden from π, but used to score π.

4.1 NESTED GENETIC ALGORITHMS

We evolve a program π by starting with a human-written symbolic regression algorithm and mutating it using large language models (LLMs) prompted with a program π together with feedback on π's performance under fitness $J(\pi, \mathcal{C}_{\text{TRAIN}})$, in the spirit of AlphaEvolve (Novikov et al., 2025) and others (Cemri et al., 2026; Sharma, 2025). This gives a nested genetic algorithm whose outer genetic search ("meta-evolution") explores different search programs while the inner genetic search uses that program to find equations.

We implement a custom LLM-evolution framework from scratch with several adaptations specific to evolving a symbolic regression algorithm (Figure 1). We maintain a population of offspring consisting of a MAP-elites population with a cell for each train task, following Agrawal et al. (2026). Each offspring π consists of a tuple of operator implementations, such as mutation, selection, survival, and loss function. Offspring are generated by selecting a parent uniformly from the population, choosing a random operator to mutate in the parent, applying one of four meta-mutation strategies—*explore* novel solutions, *refine* existing solutions, *crossover* two existing operators, or *simplify* an operator—and evaluating under eq. (2). Ultimately, our goal is to synthesize a program that is both performant and simple, so we conclude evolution with a simplification phase of evolution, during which offspring are sampled using the simplification strategy alone, and the task-niche population is replaced with a population tracking the Pareto frontier of performance versus lines of code. Further methodological details are in Appendix A.

4.2 WINNER'S CURSE AND REEVALUATION STRATEGIES

A naive strategy for estimating the expected performance of a program π is to evaluate the expectation in eq. (2) via a fixed number of Monte Carlo samples N, i.e. run π N times on each problem and average the results. Running more seeds reduces variance, but at steep computational cost that could be allocated to exploring new programs. In practice, scoring programs is far more expensive than querying LLMs to make new programs. With the few seeds necessary to efficiently cover the search space, the resulting Monte Carlo estimates of J can be high-variance.

High-variance estimates of J introduce selection bias during and after evolution, also known as *winner's curse*. More specifically, stochasticity of estimating J hinders the performance of evolution in two ways:

1. Post-evolution winner's curse: the final selected program π chosen after meta-evolution finishes will have fitness estimates biased upwards.

2. Parent selection winner's curse: During the course of evolution, lucky but low fitness offspring will be maintained in the population and selected as parents for mutation of new offspring more often, slowing the progress of the evolutionary loop.

In our setting (measuring fitness via an average of Bernoulli outcomes), evaluation uncertainty is well-approximated as normally distributed with zero mean and constant variance σ^2. In the worst case, when selecting the highest estimated fitness among K equal candidates, the expected winner's curse theoretically scales as $\mathcal{O}\left(\sigma\sqrt{\log K}\right)$. Empirically, we find winner's curse strongly affects meta-evolution performance, as shown in Figure 5.

Both forms of winner's curse can be mitigated via strategies for allocating additional evaluations to reduce regret. Formally, they are instantiations of fixed-budget Bayesian multi-arm bandit ranking and selection problems: Given a set of arms (offspring) A_i, each with N_i evaluations and posterior over true fitness $f_i \sim \mathcal{N}(\mu_i, \sigma^2/N_i)$, with fixed σ, the objective is to allocate additional evaluations m_i under budget $M, \sum m_i = M$ to maximize the expected true fitness of (1) the final best operator after evolution, or (2) the selected parent during evolution.

Because post-evolution winner's curse is only encountered once, we can afford a larger evaluation budget and do not need an efficient approach. For simplicity, we take the population at the end of evolution, discard previous evaluations to ameliorate winner's curse, reevaluate each candidate over $N = 10$ seeds, and choose the candidate with the highest score.

During evolution, evaluation strategies for dealing with parent selection winner's curse are more cost-sensitive, as the number of offspring sampled over the course of evolution is large compared to population size. We propose the following two reevaluation strategies. Both evaluate initial offspring for N_{init} seeds. **Uniform population reevaluation** reevaluates offspring that score well enough to enter the population with additional N_{reeval} seeds. **Top-two Thompson sampling (TTTS)** is a Bayesian exploration strategy for best arm identification. It allocates an evaluation budget M among the new and existing offspring using a variant of Thompson sampling. Whereas uniform reevaluation improves estimates across the population, TTTS targets best-arm identification and has asymptotic optimality guarantees under standard assumptions (Russo, 2020). Effective parent selection lies between these objectives, making their relative performance an empirical question.

4.3 Generalization of learned algorithms

Transfer to new domains. The high cost of evaluation during meta-evolution necessitates evolving over a small set of training tasks and for a limited search budget B, to allow exploring many programs π. In the best case, our synthesized symbolic regression program π transfers out of the box to new domains. We can also use the meta-evolution synthesis algorithm to *fine-tune* a symbolic regressor to a specific domain. Fine-tuning a program on a new domain means running meta-evolution on a new set of training tasks, or in the limit, only a single task from the new domain.

Extrapolating to harder search problems. Scaling search to large budgets is necessary for tackling the hardest problems of scientific discovery, but meta-evolution benefits from small train-time budgets. Therefore, it is valuable to understand whether learned symbolic regressors can extrapolate to longer search budgets. Beyond simply increasing the test-time budget B, we also consider random restarts. Given train budget B_{TRAIN} and test budget B_{TEST}, we perform $\lfloor \frac{B_{\text{TEST}}}{B_{\text{TRAIN}}} \rfloor$ independent equation searches, merge the results, and return the Pareto frontier under $f_{\mathcal{D}}$:

$$S_i \sim \pi(\mathcal{E}, \mathcal{D}, f, B_{\text{TRAIN}}), \text{ where } 1 \leq i \leq \left\lfloor \frac{B_{\text{TEST}}}{B_{\text{TRAIN}}} \right\rfloor \qquad S = \underset{e \in \bigcup_i S_i}{\arg\min} f_{\mathcal{D}}(e). \tag{3}$$

5 Results

We study our method across benchmarks with the goal of understanding (1) its ability to discover performant search algorithms from minimal starter code or (2) from state-of-the-art symbolic regression code, and the extent to which it generalizes across objectives, datasets, and search horizon.

Table 1: Results on SRBench, averaged over 10 seeds. Ground-truth recoveries are averaged over four noise levels (0, 0.001, 0.01, 0.1) and 130 tasks. Black box R^2 is averaged over 122 tasks. Improved versions of PySR and BasicSR are evaluated with a 90 second timeout.

Model	Training metric	Black-box average R^2	Ground-truth average recovery rate (%)
PySR		0.814	50.6
Evolved PySR	R^2	0.790	45.4
	GT-R^2	0.811	52.5
	GT	0.659	**58.6**
BasicSR		-0.148	19.5
Evolved BasicSR	R^2	0.742	46.0
	GT-R^2	0.817	50.8
	GT	0.715	39.3
HPO applied to PySR	R^2	0.814	47.7
	GT-R^2	**0.823**	50.6
	GT	0.813	50.8
Autoresearch applied to PySR	GT	0.810	52.9
MDLFormer	GT	–	40.5

5.1 Experimental setup

Datasets. We evolve and evaluate the discovered SR algorithms on tasks from SRBench (La Cava et al., 2021). SRBench evaluates ground truth equation recovery on a set of 130 tasks collected from the Feynman (Udrescu & Tegmark, 2020) and Strogatz (Strogatz, 2014) problem sets, and evaluates black box performance (comparing R^2, equation size, and search time) on a set of 122 machine learning tasks with no ground truth equation from PMLB (Romano et al., 2021).

Both settings evaluate with 10 random seeds while the GT setting also averages over four noise levels (0, 0.1, 0.01, and 0.001). To evolve our algorithms, we use a training set of 20 tasks from SRBench GT problems and 20 validation tasks. We evaluate our algorithms for up to 1 million max evaluations, and also compare for a fixed time budget of 90 seconds up to 15 minutes per problem. All SR algorithms are run on a single core with no parallelism. We also evaluate on EmpiricalBench, a benchmark of nine real-world scientific equation discovery problems created by Cranmer (2023). To test extrapolation, we evaluate algorithms on one core for 1 hour using repeated restarts of runs with execution budget of 1 million evaluations for both PySR and evolved PySR, using ten random seeds per task. EmpiricalBench recovery is assessed using an LLM-based evaluator with manual validation (Appendix C).

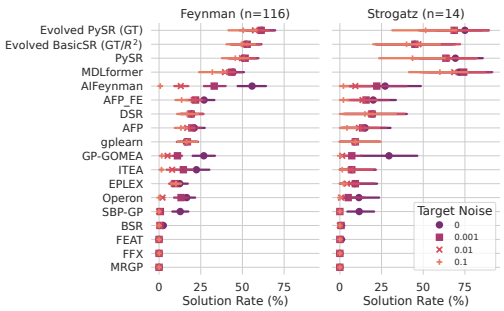


Figure 2: Recovery scores for evolved algorithms, baselines, and prior work on SRBench ground truth problems.

Baselines. We compare against baselines meant to capture the strongest representatives of different classes of approaches: PySR is the most widely used symbolic regression package; Autoresearch (Karpathy, 2026) tasks a coding agent (Claude Sonnet 4.6) with iterating on and improving PySR, sharing the evaluation harness but replacing meta-evolution with open-ended agentic code generation. We ran autoresearch to improve PySR, autonomously conducting 60 experiments over 14 hours. Hyperparameter Optimization (HPO) uses Bayesian optimization using Optuna (Akiba et al., 2019) to tune PySR's hyperparameters under the same objectives we use, with the number of trials matched to the number of evaluations used by evolution. MDLFormer (Yu et al., 2025) is a hybrid transformer+search approach which to our knowledge is the prior state-of-the-art method for SRBench ground truth domain. We also compare with published numbers for a very wide range of existing symbolic regression algorithms evaluated on SRBench.

Evaluation metrics. We consider two metrics for judging the quality of a solution to a symbolic regression problem. The **ground truth (GT)** metric checks each equation on the Pareto frontier, returning 1 if at least one equation algebraically matches a reference ground truth equation and 0 otherwise. The **black-box R^2 metric** measures the R^2 coefficient of determination between the equation's predictions and its target outputs on holdout data. To measure the ability to discover equations with large R^2 and low complexity while remaining agnostic to the correct complexity, fitness is calculated as the average R^2 across expressions of complexity 1 to the max size along the Pareto frontier. We also consider a blended metric, **GT-R^2**, which gives a fitness of 1 if the ground truth equation is recovered, and returns the R^2 fitness otherwise.

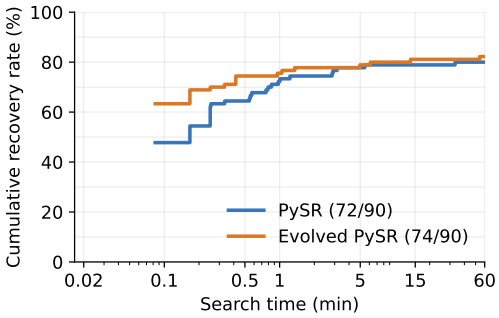


Figure 3: EmpiricalBench cumulative equation recovery rates for the evolved and original PySR, using random restarts of 1 million evaluations to extrapolate to 1 hour search time.

Table 2: SRBench ground-truth recovery rates (%) across 130 tasks and 10 random seeds

	PySR	Evolved PySR
1 million evaluations	47.4	56.1
90 seconds	50.6	58.6
15 minutes	55.4	62.1
15 minutes + restarts	60.6	63.9
Excluding train & val	47.0	51.5

Evolution from the basics versus evolution from the state of the art. Should we initialize meta-evolution with a state-of-the-art SR algorithm or should we expect our approach to work better when it begins virtually from scratch? Standing on the shoulders of the state of the art would seem to offer an advantage, but in principle could also trap meta-evolution in a deep local optimum. To answer this question we compare evolution from PySR (Cranmer, 2023), a state-of-the-art evolutionary symbolic regression algorithm, with evolution from a basic genetic algorithm skeleton which we call 'BasicSR.' BasicSR was produced by using coding agents to implement a flexible yet bare bones skeleton where all algorithmic decisions beyond the basic evolutionary loop are replaced with placeholder operators. See Appendix B.2 for details.

5.2 Experimental results

Table 1 compares performance of evolved versions of BasicSR and PySR under the three objectives to HPO, Autoresearch, the base PySR and BasicSR algorithms, and MDLFormer on the SRBench black box and ground truth domains. We find that BasicSR can be evolved to match PySR's original performance when the GT-R^2 metric is used. For PySR, evolution with the GT metric dramatically improves performance on GT tasks. Evolving PySR does not improve black-box R^2 in these experiments. Autoresearch achieves GT improvement as well, but less than evolution. Additional comparison of evolved BasicSR (GT-R^2) and PySR (GT) ground truth scores to PySR, MDLFormer, and prior work is shown in Figure 2. Subsequently, we focus on the evolved version of PySR under the GT objective as it achieved the highest performance. Evolved PySR's recovery gains persist after excluding training and validation tasks and under longer search budgets, with and without random restarts (Table 2).

On EmpiricalBench, where search is conducted for one hour, evolved PySR solves 74/90 trials, edging out PySR (72/90) while discovering solutions faster (Figure 3).

Learned algorithm analysis. Progress evolving PySR is shown in Figure 4, and a description of the evolved operators is shown in Figure 14 in Appendix E. Performance gains are mostly driven by evolution of an improved loss function. The loss function fits an affine scale and offset to the expression's predictions before measuring NRMSE, allowing search over equation structures and reducing effort spent optimizing coefficients. The mutation operator also contributes gains with a sophisticated variant of random subtree duplication. These evolved operators have precedents: calculating fitness after an affine scaling was proposed in Keijzer (2003), and subtree duplication was used in Hoai et al. (2005). The evolved operators are refinements of these ideas, and combine

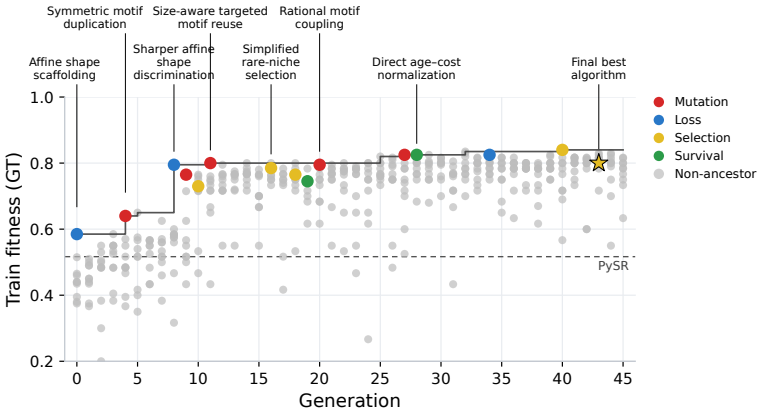


Figure 4: Evolution of PySR under the ground truth (GT) objective. Offspring each generation are shown; ancestors of the final best algorithm (starred) are colored to show the operator type that ancestor contributed to. Labels summarize the contribution of the evolved operator for selected ancestors of the final best operator.

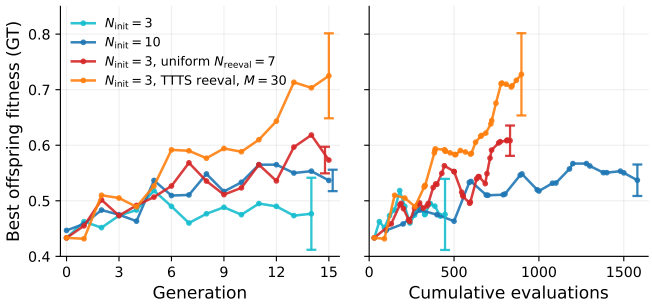


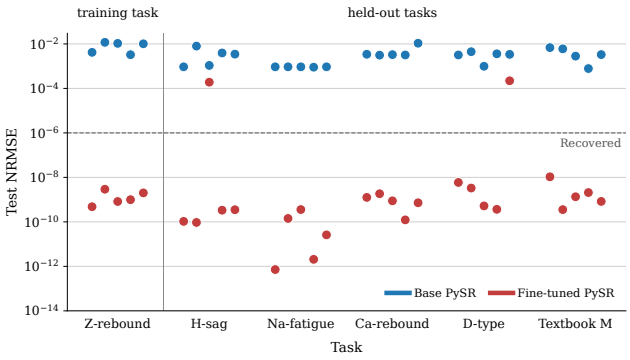
Figure 5: Reevaluation strategies compared over generations (left) or cumulative evaluations (right).

them to effectively improve PySR. Further analysis of the operators is in Appendix E. Descriptions and code of the evolved and baseline operators for PySR and BasicSR can be found in Appendices E and F. The simplification phase of evolution aids analysis here, reducing the best algorithm's operator set from 454 to 394 lines of code for PySR and from 1837 to 549 lines for BasicSR.

Reevaluation strategy ablations. Figure 5 compares meta-evolution runs using 3 or 10 initial seeds with TTTS or uniform reevaluation strategies using 3 initial seeds. Evolution with reevaluation outperforms using 3 and surprisingly even 10 initial seeds while using half the number of total evaluations. Between reevaluation strategies, TTTS outperforms uniform reevaluation. One reason TTTS can beat even 10 initial seeds is that TTTS can end up allocating many more seeds to the top candidates. Further reevaluation discussion is in Appendix D.

Fine-tuning to specialized domains. To test whether meta-evolution can fine-tune algorithms to a specialized domain, we adapt a set of six neuron equation discovery tasks from NeuronBench (Murphy, 2026). Each task is a generalized Hodgkin-Huxley model with an extra novel membrane mechanism added. With five random seeds per problem, PySR never recovers the governing equation for any of the six tasks. We found that fine-tuning PySR to improve performance on *just a single task*–running meta-evolution for 15 generations–yields ground-truth equation recovery on 23 out of 25 trials across five held-out tasks. The evolved algorithm works through a custom mutation operator specialized for discovering large additively structured equations: calculating the residual of the parent expression and proposing a child which adds a linear term with the input variable most aligned with the residual and a coefficient fit with least squares.

(a) NeuronBench



(b) MIPS

Task	Base	Evolved	MIPS-evolved
Minimum	✓	✓	✓
Maximum	✓	✓	✓
Base-3 addition	✓	✓	✓
Base-6 addition	✓	✓	✓
Alternating-last4	✓	–	✓
Base-4 addition	–	✓	✓
Base-5 addition	–	✓	✓
Base-7 addition	–	✓	✓
Parity-last4	–	–	✓
Divisibility by 7	–	–	–
Divisibility by 5	–	–	–
Alternating-last3	–	–	–
Tasks solved / 12	**5**	**7**	**9**
SR problems solved / 390	**203**	**230**	**245**

Figure 6: Fine-tuning PySR on custom domains. (a) NeuronBench. Fine-tuning PySR via meta-evolution on a single task yields an algorithm that recovers the governing equation in 23 out of 25 trials across five held-out tasks. (b) MIPS mechanistic interpretability domain, where symbolic regression is used to discover the internal mappings used by recurrent neural networks to solve algorithmic tasks.

Mechanistic interpretability transfer and fine-tuning. We apply meta-evolution to the discovery of equations that describe the internal mechanisms of recurrent neural networks (RNNs) trained on algorithmic tasks. Prior work, denoted MIPS—Mechanistic Interpretability via Program Synthesis—introduces a pipeline for distilling an RNN into code by 1) state space simplification and discretization, 2) converting into a finite state machine, and 3) applying symbolic regression, linear regression, or boolean regression to learn formulas for the internal and output mappings (Michaud et al., 2024). One task decomposes into 2–5 SR subproblems for learning the output and hidden state mappings. Michaud et al. (2024) successfully discover programs for 32 of 62 algorithmic tasks. Of the remaining 30 tasks, we found 12 with consistent regression targets, for a total of 39 SR problems (the remainder failed earlier steps of the distillation pipeline). We evaluated PySR, the improved PySR evolved on SRBench, and a version of PySR fine-tuned on the 39 unsolved SR problems and 12 SR problems solved by prior work. We ran each method over the 39 problems for 10 random seeds each, and considered a task solved if all subproblems were solved with one random seed. SRBench-evolved PySR improvements transfer to this drastically different setting, with two additional tasks and 27 additional successful SR trials, while MIPS fine-tuned PySR solves an additional two tasks and 15 additional successful trials beyond that (Figure 6(b)). This provides additional evidence that algorithms learned with meta-evolution transfer to new domains and that meta-evolution can be effectively used to fine-tune algorithms to specialized domains.

6 Conclusion and Future Work

We give a meta-evolutionary approach to learning to discover equations, yielding symbolic regressors that match or exceed human state of the art on standard benchmarks, and which can be transferred and tuned to datasets as different as mechanistic interpretability and neural spiking signals. The technical approach requires program synthesis under a stochastic quantitative objective, and we show how reevaluation strategies, particularly top-two Thompson sampling, help in this setting, a finding which could have broader applicability whenever a stochastic program must be optimized. Our approach has limitations. Meta-evolution is computationally expensive, search improvements obtained in training do not always persist on held-out tasks, and while we found LLM-evolution to be more effective at synthesizing an improved SR algorithm than "autoresearch" approaches, continued increases in LLM capabilities may alter this rank-ordering. Designing approaches that combine strengths of LLM-evolution with the potential of autoresearch will likely be an important future direction. Last, it is important to acknowledge that the evolved algorithms are not necessarily new discoveries by LLMs. There is a vast literature of techniques for genetic algorithms which the LLMs are likely harnessing to propose improvements.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods and experiments behind the present work. Specifically, we relied on coding agents heavily. Manual implementation and code review were undertaken. Coding agents were used throughout the research project for implementation of methods, experiments, analysis, experimental design, and experimental interpretation. We also used coding agents to help make figures and tables, and to assist with figure captions and descriptions of methods. We strove to write the paper manually, with generative AI used only for editing. However, parts of the appendix were written with the assistance of coding agents to document experimental details.

Coding agents methodologically contributed to the experiments in two ways. First, the BasicSR baseline was implemented by coding agents; see Appendix B.2 for details. Second, the manual evaluation of Pareto frontiers to check for equation recovery for EmpiricalBench was replaced by evaluation using LLM evaluators; see Appendix C for details.

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alex Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=RQm2KQTM5r.

Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=rkE3y85ee.

Luca Biggio, Tommaso Bendinelli, Alexander Neitz, Aurelien Lucchi, and Giambattista Parascandolo. Neural symbolic regression that scales. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 936–945. PMLR, 2021. URL https://proceedings.mlr.press/v139/biggio21a.html.

Rudy Bunel, Alban Desmaison, M. Pawan Kumar, Philip H. S. Torr, and Pushmeet Kohli. Learning to superoptimize programs. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=r1rz6U5lg.

Mert Cemri, Shubham Agrawal, Akshat Gupta, Shu Liu, Audrey Cheng, Qiuyang Mang, Ashwin Naren, Lutfi Eren Erdogan, Koushik Sen, Matei Zaharia, Alex Dimakis, and Ion Stoica. AdaEvolve: Adaptive LLM driven zeroth-order optimization, 2026. URL https://arxiv.org/abs/2602.20133.

Miles Cranmer. Interpretable machine learning for science with PySR and SymbolicRegression.jl, 2023. URL https://arxiv.org/abs/2305.01582.

F. O. de Franca, M. Virgolin, M. Kommenda, M. S. Majumder, M. Cranmer, G. Espada, L. Ingelse, A. Fonseca, M. Landajuela, B. Petersen, R. Glatt, N. Mundhenk, C. S. Lee, J. D. Hochhalter, D. L. Randall, P. Kamienny, H. Zhang, G. Dick, A. Simon, B. Burlacu, Jaan Kasak, Meera Machado, Casper Wilstrup, and W. G. La Cava. SRBench++: Principled benchmarking of symbolic regression with domain-expert interpretation. *IEEE Transactions on Evolutionary Computation*, 29(4): 1127–1137, 2025. doi: 10.1109/TEVC.2024.3423681.

Andrew Dotson, Zaki Panjsheeri, Anusha Reddy Singireddy, Douglas Q. Adams, Emmanuel Ortiz-Pacheco, Marija Cuic, Yaohang Li, Huey-Wen Lin, Simonetta Liuti, Matthew D. Sievert, Marie

Boer, Gia-Wei Chern, Michael Engelhardt, and Gary R. Goldstein. Generalized parton distributions from symbolic regression, 2025. URL https://arxiv.org/abs/2504.13289.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pp. 835–850. Association for Computing Machinery, 2021. doi: 10.1145/3453483.3454080. URL https://doi.org/10.1145/3453483.3454080.

Kanishk Gandhi, Denise Lee, Gabriel Grand, Muxin Liu, Winson Cheng, Archit Sharma, and Noah D. Goodman. Stream of search (SoS): Learning to search in language. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=2cop2jmQVL.

Arya Grayeli, Atharva Sehgal, Omar Costilla-Reyes, Miles Cranmer, and Swarat Chaudhuri. Symbolic regression with a learned concept library. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/4ec3ddc465c6d650c9c419fb91f1c00a-Abstract-Conference.html.

Nguyen Xuan Hoai, Robert Ian McKay, Daryl Essam, and Hoang Tuan Hao. Genetic transposition in tree-adjoining grammar guided genetic programming: The duplication operator. In *Genetic Programming: Proceedings of EuroGP 2005*, volume 3447 of *Lecture Notes in Computer Science*, pp. 108–119. Springer, 2005. doi: 10.1007/978-3-540-31989-4_10. URL https://doi.org/10.1007/978-3-540-31989-4_10.

Samuel Holt, Zhaozhi Qian, and Mihaela van der Schaar. Deep generative symbolic regression. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=o7koEEMA1bR.

Yaochu Jin and Jürgen Branke. Evolutionary optimization in uncertain environments—a survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317, 2005. doi: 10.1109/TEVC.2005.846356. URL https://doi.org/10.1109/TEVC.2005.846356.

Pierre-Alexandre Kamienny, Stéphane d'Ascoli, Guillaume Lample, and François Charton. End-to-end symbolic regression with transformers. In *Advances in Neural Information Processing Systems*, volume 35, pp. 10269–10281, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/hash/42eb37cdbefd7abae0835f4b67548c39-Abstract-Conference.html.

Pierre-Alexandre Kamienny, Guillaume Lample, Sylvain Lamprier, and Marco Virgolin. Deep generative symbolic regression with Monte-Carlo-Tree-Search. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 15655–15668. PMLR, 2023. URL https://proceedings.mlr.press/v202/kamienny23a.html.

Andrej Karpathy. autoresearch. GitHub, 2026. URL https://github.com/karpathy/autoresearch. Accessed September 23, 2026.

Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *Genetic Programming: Proceedings of EuroGP 2003*, volume 2610 of *Lecture Notes in Computer Science*, pp. 70–82. Springer, 2003. doi: 10.1007/3-540-36599-0_7. URL https://doi.org/10.1007/3-540-36599-0_7.

Johannes Kepler. *New Astronomy*. Cambridge University Press, Cambridge, 1609. Originally published as *Astronomia Nova*.

William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio de Franca, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason Moore. Contemporary symbolic regression methods and their relative performance. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper_files/paper/2021/hash/c0c7c76d30bd3dcaefc96f40275bdc0a-Abstract-round1.html.

Pat Langley. Data-driven discovery of physical laws. *Cognitive Science*, 5(1):31–54, 1981. doi: 10.1111/j.1551-6708.1981.tb00869.x.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 19(2):189–223, 2011. doi: 10.1162/EVCO_a_00025.

Li Li, Minjie Fan, Rishabh Singh, and Patrick Riley. Neural-guided symbolic regression with asymptotic constraints. In *NeurIPS Workshop on Knowledge Representation and Reasoning Meets Machine Learning*, 2019. URL https://kr2ml.github.io/2019/papers/KR2ML_2019_paper_7.pdf.

Shu Liu, Shubham Agarwal, Monishwaran Maheswaran, Mert Cemri, Zhifei Li, Qiuyang Mang, Ashwin Naren, Ethan Boneh, Audrey Cheng, Melissa Z. Pan, Alexander Du, Kurt Keutzer, Alvin Cheung, Alexandros G. Dimakis, Koushik Sen, Matei Zaharia, and Ion Stoica. EvoX: Meta-evolution for automated discovery, 2026. URL https://arxiv.org/abs/2602.23413.

Eric J. Michaud, Isaac Liao, Vedang Lad, Ziming Liu, Anish Mudide, Chloe Loughridge, Zifan Carl Guo, Tara Rezaei Kheirkhah, Mateja Vukelić, and Max Tegmark. Opening the ai black box: program synthesis via mechanistic interpretability, 2024. URL https://arxiv.org/abs/2402.05110.

Terrell Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P. Santiago, Daniel Faissol, and Brenden K. Petersen. Symbolic regression via deep reinforcement learning enhanced genetic programming seeding. In *Advances in Neural Information Processing Systems*, volume 34, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/d073bb8d0c47f317dd39de9c9f004e9d-Abstract.html.

Kevin Murphy. Model discovery agent: LLM-assisted bayesian experiment design for data-efficient discovery of mechanistic world models, 2026. URL https://arxiv.org/abs/2608.09696.

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

Mihai Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005. doi: 10.1162/1063656054794815. URL https://doi.org/10.1162/1063656054794815.

Brenden K. Petersen, Mikel Landajuela, T. Nathan Mundhenk, Claudio P. Santiago, Soo K. Kim, and Joanne T. Kim. Deep symbolic regression: Recovering mathematical expressions from data via risk-seeking policy gradients. In *International Conference on Learning Representations*, 2021. URL https://openreview.net/forum?id=m5Qsh0kBQG.

Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu.com, 2008. URL http://www.gp-field-guide.org.uk. With contributions by J. R. Koza.

Joseph D Romano, Trang T Le, William La Cava, John T Gregg, Daniel J Goldberg, Praneel Chakraborty, Natasha L Ray, Daniel Himmelstein, Weixuan Fu, and Jason H Moore. Pmlb v1.0: an open source dataset collection for benchmarking machine learning methods. *arXiv preprint arXiv:2012.00058v2*, 2021.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://www.nature.com/articles/s41586-023-06924-6.

Daniel Russo. Simple Bayesian algorithms for best-arm identification. *Operations Research*, 68 (6):1625–1647, 2020. doi: 10.1287/opre.2019.1911. URL https://doi.org/10.1287/opre.2019.1911.

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

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K. Reddy. LLM-SR: Scientific equation discovery via programming with large language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=m2nmp8P5in.

Steven H. Strogatz. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry, and Engineering*. Westview Press, second edition, 2014. ISBN 9780813349107.

Silviu-Marian Udrescu and Max Tegmark. AI Feynman: A physics-inspired method for symbolic regression. *Science Advances*, 6(16):eaay2631, 2020. doi: 10.1126/sciadv.aay2631. URL https://doi.org/10.1126/sciadv.aay2631.

Niki van Stein and Thomas Bäck. LLaMEA: A large language model evolutionary algorithm for automatically generating metaheuristics. *IEEE Transactions on Evolutionary Computation*, 29(2): 331–345, 2025. doi: 10.1109/TEVC.2024.3497793. URL https://doi.org/10.1109/TEVC.2024.3497793.

Zihan Yu, Jingtao Ding, Yong Li, and Depeng Jin. Symbolic regression via mdlformer-guided search: from minimizing prediction error to minimizing description length. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL https://openreview.net/forum?id=ljAS7cPAU0.

Mert Yuksekgonul, Daniel Koceja, Xinhao Li, Federico Bianchi, Jed McCaleb, Xiaolong Wang, Jan Kautz, Yejin Choi, James Zou, Carlos Guestrin, and Yu Sun. Learning to discover at test time, 2026. URL https://arxiv.org/abs/2601.16175.

A META-EVOLUTION ALGORITHMIC DETAILS

A.1 META-EVOLUTION ALGORITHM

Representation. Each meta-evolution individual is a set of SR operator implementations, such as mutation, loss, survival, and selection for PySR.

Generation structure. Each generation proposes a fixed number of offspring. An offspring inherits all operators from its selected parent, then replaces one operator with a mutated variant.

Meta-mutation. We choose randomly between four mutation modes: explore, refine, simplify, and crossover. The explore prompt encourages novel solutions with no parent selected. The refine prompt encourages making small changes to produce an improved variant. The simplify prompt asks to simplify an operator (removing for example special cases, branches, or heuristics), and crossover asks to create a new operator incorporating ideas from two parent operators.

Initial population. We generate an initial population of offspring by sampling the explore operator to generate variants of the base algorithm.

Execution feedback (PySR evolution only). For the parent, we find a task and evaluation seed that the parent performed poorly on and show the execution trace. The execution trace shows the Pareto frontier three times over the course of evolution (at 33%, 66%, and 100% progress), and the target equation.

SR reference. We create a reference file for each operator type showing relevant API functions, usage guidance, examples of valid operators, and operator-specific instructions.

Data-aware mutations (PySR evolution only). the mutation operator, with 50% probability, the problem data is available as an argument to mutation to enable exploring "data-aware" smart mutation strategies.

Code validation. Generated operator code is loaded into Julia and tested on synthetic inputs.

Evaluation. Evaluation is conducted synchronously in parallel on a Slurm cluster for N random seeds $\times$ O offspring $\times$ T train tasks total symbolic regression search problems.

Reevaluation strategies. Reevaluation strategies allocate additional evaluations to selected offspring and are conducted in parallel with the evaluation of the next generation's offspring.

Population strategies. By default, the population is a MAP-Elites population with a cell for performance on each task in the training set. We also experimented using a population strategy of the top P offspring by fitness.

Final candidate identification. 10 seeds are evaluated on the final top P offspring on the validation set, and the top operator is selected.

Simplification phase. We evolve only using the simplify meta-mutation strategy and we switch to a complexity-loss Pareto frontier population. We measure the total lines of code across each offspring's operator implementations and divide the observed range into P equal-width budgets. Then we create a Pareto frontier across those buckets. For parent selection, we select parents uniformly over the population.

Pseudocode for the meta-evolution algorithm is shown in Algorithm 1.

Algorithm 1 Meta-evolution

Inputs: `baseline`; G_{total} total generations; G_{simplify} simplification generations; O offspring; P population size; N_{init} initial evaluation seeds; N_{final} final reevaluation seeds; `tasks` training tasks; `reeval_strategy` reevaluation policy
Evaluate: mean task score over the requested number of fresh random seeds.

`population` $\leftarrow$ [`baseline`]
$\hat{J}, N \leftarrow$ DICT(), DICT() # Fitness estimates and seed counts by program
$\hat{J}$[`baseline`] $\leftarrow$ EVALUATE(`baseline`, `tasks`, N_{init}); N[`baseline`] $\leftarrow N_{\text{init}}$
`pending` $\leftarrow \varnothing$ # Pairs (π, m_π) scheduled in the previous generation
for $g = 0, \ldots, G_{\text{total}} - 1$ **do**
  `offspring` $\leftarrow$ []
    # Generate offspring
  for $o = 1, \ldots, O$ **do**
    `sr_operator` $\leftarrow$ RANDOMCHOICE([`mutation`, `survival`, `selection`, `loss`])
    if $g = 0$ **then**
      `mode` $\leftarrow$ `explore`
    else if $g \geq G_{\text{total}} - G_{\text{simplify}}$ **then**
      `mode` $\leftarrow$ `simplify`
    else
      `mode` $\leftarrow$ RANDOMCHOICE([`explore`, `refine`, `crossover`, `simplify`])
    end if
    `parent1`, `parent2` $\sim$ TOURNAMENTSELECTION(`population`)
    `prompt` $\sim$ GENERATEPROMPT(`tasks`, g, `parent1`, `parent2`)
    `child` $\sim$ LLM(`prompt`)
    `offspring`.APPEND(`child`)
  end for
  # Initial evaluations of new offspring; execute the previous reevaluation schedule
  for $\pi \in$ `offspring` **do**
    $\hat{J}[\pi] \leftarrow$ EVALUATE(π, `tasks`, N_{init}); $N[\pi] \leftarrow N_{\text{init}}$
  end for
  for $(\pi, m_\pi) \in$ `pending` **do**
    $\bar{J} \leftarrow$ EVALUATE(π, `tasks`, m_π)
    $\hat{J}[\pi] \leftarrow \dfrac{N[\pi]\hat{J}[\pi] + m_\pi \bar{J}}{N[\pi] + m_\pi}$; $N[\pi] \leftarrow N[\pi] + m_\pi$
  end for
  if $g \geq G_{\text{total}} - G_{\text{simplify}}$ **then**
    Set population to the Pareto frontier of score versus LOC.
  else
    Set population to the set of programs with maximum score on at least one task.
  end if
  Fill remainder of population with individuals selected by highest average task score, up to size P.
  if $g < G_{\text{total}} - 1$ **then**
    `pending` $\leftarrow$ `reeval_strategy`($\hat{J}, N$) # Store (π, m_π) pairs for next generation
  end if
end for
  # Final reevaluation: winner's curse
$\forall \pi \in$ `population` : $\hat{J}[\pi] \leftarrow$ EVALUATE(π, `tasks`, N_{final})
return $\arg\max_{\pi \in \texttt{population}} \hat{J}[\pi]$

Meta-evolution hyperparameters. We evolve for 30 generations with 10 offspring per generation. For a reevaluation strategy we use uniform reevaluation with $N_{\text{init}} = 3$ and $N_{\text{reeval}} = 7$. Final operator selection is chosen from the population based on validation score over 10 random seeds. For the evolved algorithms which we interpret, we add additional simplification-only generation until plateau is qualitatively reached: 15 additional generations for PySR evolved under GT objective, and 60 additional generations for BasicSR evolved under GT-R^2. We sample offspring from an ensemble of GPT 5.6-Sol, Claude Opus 5, Claude Sonnet 5, and Gemini 3.7 using medium reasoning level. Evolution cost roughly \$0.25–\$0.75 per generation.

B PySR and BasicSR baseline details

B.1 PySR evolution

PySR's search backend is implemented in the `SymbolicRegression.jl` Julia library containing over 10,000 lines of code across dozens of files. To make evolution tractable, we chose to evolve four components of PySR: the mutation, selection, survival, and loss evaluation functions. We extended the backend with runtime dispatch hooks that allow custom Julia implementations of these components to be dynamically loaded and executed within the search algorithm, making use of Julia's runtime code evaluation and just-in-time compilation abilities. PySR samples mutations from a weighted mixture of mutation types; we include the evolvable custom mutation operator with a weight corresponding to an approximately 5% chance of being sampled. The evolved selection, survival, and loss functions replace their default counterparts. LLM-generated Julia implementations of the operators are validated using operator-specific execution tasks before being evaluated by running the full search algorithm on each task in the training set. The generated operators operate directly on native Julia trees, populations, and datasets.

B.2 BasicSR evolution

In our implementation of BasicSR, we sought to implement a flexible genetic algorithm skeleton that can express diverse approaches to search, as well as initialize the skeleton with basic implementation decisions to start. In particular, a guiding principle to implementing BasicSR and its configuration of evolvable operator slots was that an algorithm similar to PySR could in theory be expressed by an evolved configuration. The goal was *not* to create a basic genetic algorithm implementation for SR that does not have any trace of prior work. Much of the structure of BasicSR shares similarities to the PySR search algorithm, even if it, upon initialization, lacks the sophisticated operators PySR implements.

To construct a BasicSR implementation, we followed the following recipe, using coding agents heavily to complete the implementation: At each step, we evaluated the resulting skeleton to ensure that, when configured with PySR components, performance was comparable to PySR.

1. Implement a single-file version of PySR that recreates the core search algorithm, with bells and whistles that are not used in our setting (such as expression templates, exporting equations to PyTorch, and parallelization of search) removed to simplify the implementation.
2. Refactor the single-file PySR implementation into two components: a `SRSkeleton.jl` file and a `PySRConfig.jl` file. The configuration file separates out implementation of the SR operator functions we evolve in BasicSR: mutation, loss, selection, survival, acceptance, update state, update population.
3. Implement a BasicSR configuration file with minimal versions of the operators.

Pseudocode of the BasicSR skeleton is shown in Figure 7. The original and evolved components of BasicSR are shown in Figure 15.

C Domain data and evaluation details

C.1 SRBench

We evaluate symbolic regression on the 130 ground-truth problems in SRBench, comprising 116 Feynman problems and 14 Strogatz problems. For each trial, we sample at most 1,000 observations and use a seeded $80/20$ training–validation split. Noise experiments add Gaussian noise to the training targets with standard deviation equal to a specified fraction of their root-mean-square magnitude, using noise levels 0, 0.001, 0.01, and 0.1; validation targets remain unperturbed. We assess equation recovery across the complete discovered Pareto frontier. A trial is successful if at least one expression passes the validation-R^2 threshold of 0.5 and matches the reference equation under the symbolic checker's allowance for an additive or multiplicative constant. Recovery rates are averaged over problems and stochastic trials. We retain all 130 problems in the evaluation denominator.

```
function FitSkeletonSR(X, y, config)
    policy_state ← init_state(config)
    populations ← 15 populations of 33 small random trees
    UPDATE_STATE!(populations)
    for iteration = 1 . . . 100 do
        for each population P do
            for 1 . . . 380 do
                UPDATE_STATE!(populations)
                for 1 . . . 3 do
                    if rand ≤ 0.0259 then
                        a ← SELECTION(P);  b ← SELECTION(P)    (distinct)
                        children ← CROSSOVER(a, b)
                        Compute LOSS_FUNCTION(child) for each child
                        Keep children passing ACCEPTANCE(parent, child)
                        if children non-empty then
                            P ← SURVIVAL(P, children)
                        end if
                    else
                        parent ← SELECTION(P)
                        child ← MUTATION(parent)
                        Compute LOSS_FUNCTION(child)
                        if ACCEPTANCE(parent, child) then
                            P ← SURVIVAL(P, [child])
                        end if
                    end if
                end for
            end for
            UPDATE_STATE!(populations)
            populations ← UPDATE_POPULATION(populations)
        end for
    end for
    return policy_state.archive
end function
```

Figure 7: BasicSR algorithm skeleton pseudocode. The *policy* (in BLUE) is the evolvable configuration of the BasicSR skeleton, consisting of operators for calculating LOSS_FUNCTION, SELECTION, MUTATION, CROSSOVER, ACCEPTANCE, SURVIVAL, UPDATE_POPULATION, and UPDATE_STATE!. The overall skeleton structure and constants (population sizes, number of iterations, etc.) originate from the PySR algorithm implementation. BasicSR operator implementations and evolved operators are shown in Figure 15.

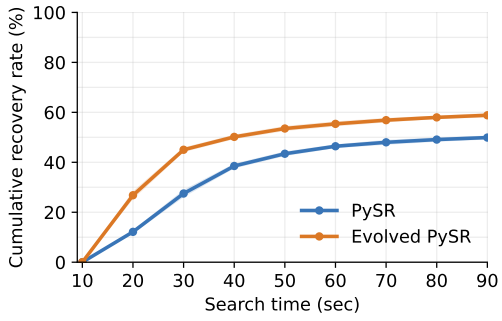


Figure 8: Cumulative equation recovery rates for the evolved and original PySR algorithms on SRBench ground truth problems.

Table 3: Results on EmpiricalBench across 10 random seeds. Large numbers show number of equation recoveries, while smaller numbers show the number of exact, near, and missed recoveries. The evolved PySR algorithm, trained on tasks from SRBench, generalizes to real world scientific discovery problems, matching PySR performance across the board while slightly improving on the Rydberg problem.

	PySR	Evolved PySR
Hubble	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Kepler	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Newton	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Planck	**0/10** (0, 10, 0)	**0/10** (0, 10, 0)
Leavitt	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Schechter	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Bode	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Ideal Gas	**10/10** (10, 0, 0)	**10/10** (10, 0, 0)
Rydberg	**2/10** (2, 7, 1)	**4/10** (4, 6, 0)

C.2 EMPIRICALBENCH.

EmpiricalBench contains nine historical empirical relationships: Hubble, Kepler, Newton, Bode, Leavitt, Schechter, ideal gas, Planck, and Rydberg. The datasets preserve the benchmark's observation noise and target transformations while inputs retain their original scales. Each search uses all available observations for fitting and assessment, reflecting the objective of recovering the underlying relationship from a fixed dataset. Recovery is assessed across the discovered frontier using dataset-specific reference families: fitted constants are permitted where the reference allows them, while fixed powers and relative coefficients must match.

The original PySR paper (Cranmer, 2023) evaluates performance on EmpiricalBench using manual review of the equations returned by each method to check for recovering the equation, due to inaccuracies from automated checkers like those used for SRBench. However, manual review is very time-intensive and is prone to mistakes. In addition, it does not scale to reviewing many methods over many timesteps, for example our plot of solve rate over time comparing PySR to the evolved PySR would be prohibitive to assess manually. For our evaluation on EmpiricalBench, we replaced manual review with LLM review. We prompted GPT-5.6-Terra with medium reasoning to look at each equation in the Pareto frontier and determine whether the equation is an exact match to the target equation, close match, or miss, making an independent LLM request for each equation in the Pareto frontier. Exact recovery requires having the same functional family as the target equation, up to algebraic equivalence and free fitted constants (fixed powers and relative coefficients must be preserved).

Manual review of selected evaluations was used during development to ensure accuracy. To validate this approach, we compared to the manual review of PySR reported in the PySR paper. Our LLM-based review differs only in two Rydberg seeds our LLM-review marked as exact, which were manually verified to be legitimate recoveries.

Table 3 gives a full breakdown of PySR versus evolved PySR performance on EmpiricalBench. Most tasks are saturated and solved quickly, and Rydberg is the only task where a difference between the two methods is found, with a slightly improved recovery rate for evolved PySR.

C.3 MIPS.

Mechanistic-Interpretability-based Program Synthesis (MIPS) (Michaud et al., 2024) extracts executable programs from recurrent neural networks trained on algorithmic tasks. The original pipeline searches for small, accurate networks, simplifies their representations, and uses Boolean or integer autoencoders to identify discrete hidden states. Regression (Linear, boolean, or symbolic) then recovers symbolic expressions for the state-transition and output functions, which are assembled into Python programs. The authors report successful synthesis on 32 of 62 algorithmic tasks. We continue from the released pretrained networks and integer-state extraction pipeline, focusing on improving the symbolic-regression stage. Each network yields one scalar regression problem per hidden-state coordinate and output coordinate: predicting the next state from the previous state and current input, or predicting the output from the current state. Thus, one algorithmic task can contribute two or more symbolic regression problems to solve in order to solve the task.

Refining the state discretization. Of the 30 tasks reported as unsolved by Michaud et al. (2024), eight have deterministic state-transition and output relations under our reproduction, making them candidates for recovery through improved symbolic regression. These tasks contribute 22 scalar regression problems. The remaining 22 tasks are initially excluded: 17 have conflicting targets under the original state encoding, and five exceed the integer encoder implementation's supported hidden-state dimensionality. To expand the candidate set, we refine the state discretization to resolve encoding conflicts.

The original integer encoding rounds continuous lattice coordinates Z to $\mathrm{round}(Z)$. This can merge distinct states whose transitions or outputs differ, producing contradictory regression targets that no deterministic expression can recover exactly. We retain the learned lattice basis but refine its resolution using $\mathrm{round}(kZ)$, testing powers of four from $k = 1$ through $k = 2^{24}$. We also test an encoding that concatenates the coarse state with a quantized within-cell residual. Among the 17 tasks with conflicting original encodings, the sweep finds conflict-free representations for ten, of which six retain at most 1,024 observed states. Subsequent artifact construction and validation leave four additional tasks requiring symbolic regression: Parity-Last-4 ($k = 4$), Div-7 ($k = 16$), Div-5 ($k = 256$), and Alternating-Last-3 ($k = 2^{22}$). Of the other two compact candidates, Parity-of-Index is solved by rounded linear regression, while Balanced-Parenthesis remains conflicted in the rebuilt artifacts and is excluded. The four retained tasks have deterministic relations across the generated training and held-out data and contribute 17 additional scalar regression problems. This expands the candidate set from eight to twelve previously unsolved algorithmic tasks, comprising 39 scalar regression problems in total. Determinism makes exact relation recovery possible on the observed data but does not guarantee that symbolic regression can discover a valid solution.

Meta-training tasks and objective. We evolve PySR on 51 scalar regression problems from 17 algorithmic tasks. These comprise the 39 problems from the twelve previously unsolved tasks described above and 12 problems from five tasks reported as solved by MIPS. A scalar problem is recovered by an expression if the expression predicts each entry of its relation within absolute tolerance 10^{-9}. We evolve for 20 generations with ten offspring per generation.

For final evaluation, we take each algorithm (PySR, PySR evolved on SRBench, and PySR fine-tuned on MIPS) and run 10 seeds per SR problem. A task is considered solved if all relations are solved for at least one of the ten seeds.

Results of the tasks solved and subtasks solved for PySR, evolved PYSR, and MIPS-evolved PySR are shown in Figure 9. An example MIPS task is shown in Figure 10—the task that MIPS-evolved PySR solved but no other method did.

NeuronBench. We use a fully observable adaptation of NeuronBench containing six membrane-dynamics worlds: `z_rebound`, `h_sag`, `na_fatigue`, `ca_rebound`, `d_type`, and `textbook_M`. Each task exposes the external current, membrane voltage, and all channel open fractions, and asks the symbolic regressor to recover the membrane vector field $\mathrm{d}V/\mathrm{d}t$. This converts the benchmark into a noiseless supervised equation-discovery problem without hidden gating-state inference or active experiment selection. We generate independent scrambled-Sobol designs containing 1,024 training states and 16,384 held-out states, spanning external currents in $[-40, 20]$, voltages in $[-95, 60]$, and channel open fractions in $[0, 1]$. Targets are divided by their training-set

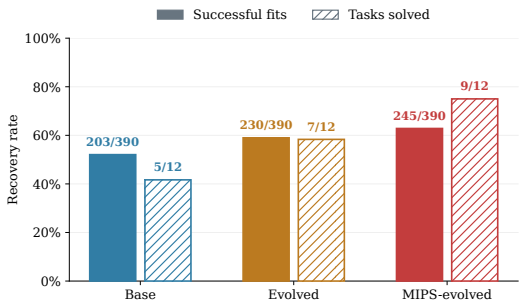


Figure 9: Full MIPS task and SR subproblem solve rate for PySR, evolved PySR, and MIPS-evolved PySR, when evaluated on 12 tasks, 39 subproblems, and 10 random seeds.

Parity-last 4

Scalar relation	Input-table size	Base	Evolved	MIPS-evolved	Discovered equation (MIPS-evolved)
$h_{t,0} = f_0(h_{t-1}, x_t)$	32 state-input pairs	0/10	0/10	**1/10**	$\min(((\max(x_1, x_0) - x_3)x_1 - x_3)x_1 - x_3 + 2,$ $4\,\lvert\mathrm{xor}(x_1(0.227249 - x_3), x_1) + x_3 - \mathrm{zero}(x_1)\rvert)$
$h_{t,1} = f_1(h_{t-1}, x_t)$	32 state-input pairs	**5/10**	**3/10**	**10/10**	$\lvert\min(\min(2.98567x_3 + 0.660363x_1,\ 2.32073)$ $+ x_3\max(3.00001, x_1),\ 9.32075)\rvert$
$h_{t,2} = f_2(h_{t-1}, x_t)$	32 state-input pairs	**10/10**	**10/10**	**10/10**	$4\,\mathrm{eq}(\mathrm{lt}(x_1, 2x_0), x_3)$
$y_t = g(h_t)$	16 states	**10/10**	**10/10**	**10/10**	$\min(1, x_2)$
Subproblems recovered across seeds		**3/4**	**3/4**	**4/4**	

Figure 10: Elaboration of the MIPS task solved by MIPS-evolved PySR but not solved by base PySR or SRBench-evolved PySR. The four SR subproblems are shown along with state space, recovery rates, and MIPS-evolved discovered equation.

root-mean-square magnitude for numerical conditioning, and the search vocabulary contains addition, subtraction, and multiplication. Recovery is determined numerically over the entire discovered frontier: a trial succeeds if any expression achieves held-out RMSE divided by the held-out target RMS of at most 10^{-6}.

D REEVALUATION STRATEGIES

D.1 TOP-TWO THOMPSON SAMPLING

Choosing which programs to reevaluate is a *best-arm identification* problem. Each program is an arm, and each extra evaluation seed is a noisy measurement of that arm's mean fitness. The goal is to identify the best arm, not to maximize the fitness observed along the way. We use top-two Thompson sampling (TTTS) (Russo, 2020), which is proven asymptotically optimal for this problem: among allocation rules that give a fraction β of measurements to the best arm, it identifies the best arm at the fastest possible rate, and with $\beta = 1/2$ its rate is within a factor of two of the optimum over all rules. TTTS refines Thompson sampling (TS), which measures each arm with the posterior probability that it is the best. TS is designed to maximize cumulative reward. As a result, once the posterior favors one arm, TS measures that arm almost exclusively, and competing arms get too few measurements to be ruled out. TTTS fixes this by giving part of the budget to the strongest challenger. At each step, TTTS:

1. Samples $\tilde{\theta}$ from the posterior and lets $I = \arg\max_i \tilde{\theta}_i$.
2. With probability β, measures arm I.
3. Otherwise, resamples until the argmax is some $J \neq I$, and measures arm J.

We model the fitness from each seed as Gaussian with a known noise level σ shared by all programs. Under a flat prior, the posterior for program i is $\mathcal{N}(\mu_i, \sigma^2/N_i)$, where μ_i is its mean fitness over N_i seeds. Instead of simulating the steps above, we compute the TTTS measurement probabilities numerically. The probability that TS picks arm i is

$$\alpha_i = \int \phi_i(x) \prod_{j \neq i} \Phi_j(x)\, dx, \tag{4}$$

where ϕ_i is the posterior density of arm i and Φ_j is the posterior CDF of arm j. We evaluate this integral with Gauss–Hermite quadrature. The TTTS probabilities then follow in closed form (Russo, 2020):

$$\psi_i = \alpha_i \Big(\beta + (1-\beta) \sum_{j \neq i} \frac{\alpha_j}{1-\alpha_j} \Big). \tag{5}$$

TTTS is sequential: it updates the posterior after every measurement. Our evaluations are expensive and run in parallel, so we use a batched version with a fixed budget of M reevaluation seeds per generation. Each generation, we compute ψ over the whole archive, sample M programs from ψ with replacement, and run the resulting seeds in parallel with the evaluation of new offspring. When they finish, we update μ_i and N_i for each reevaluated program. We use $\beta = 1/2$ and a fixed σ estimated offline from an earlier run (Figure 12).

Offline comparison of reevaluation strategies. To measure the impact of reevaluation strategies on evolution and determine the best evaluation strategy, we conducted ran evolution of PySR with $N_{\text{init}} = 10$ seeds per offspring. Treating the resulting fitness scores as oracle true fitness scores, we conducted an offline analysis of different evaluation strategies—no reevaluation with $N_{\text{init}} < 10$, uniform reevaluation of N_{reeval} seeds for new population members, and top-two Thompson sampling (TTTS) with varying reevaluation budgets M. Each reevaluation strategy determines which seeds are revealed, up to 10 seeds per offspring, and selects parents based off estimated fitness using $n = 2$ tournament selection over the population. The expected true parent fitness (computed from 10 seeds) is compared as a function of evaluations. Figure 11(a) shows that TTTS and uniform reeval improve expected parent fitness while conducting fewer evaluations than using initial evaluations alone.

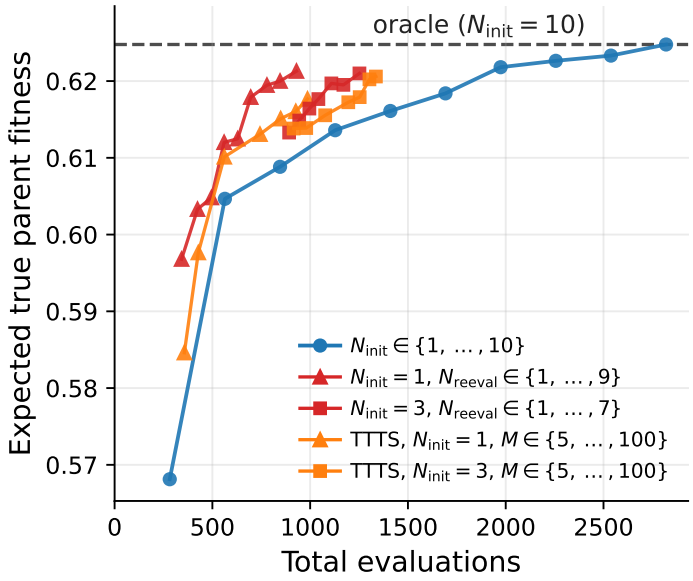


Figure 11: Offline comparison of fixed evaluation, uniform reevaluation, and top-two Thompson sampling (TTTS), showing expected true parent fitness of the selected parent versus total evaluations.

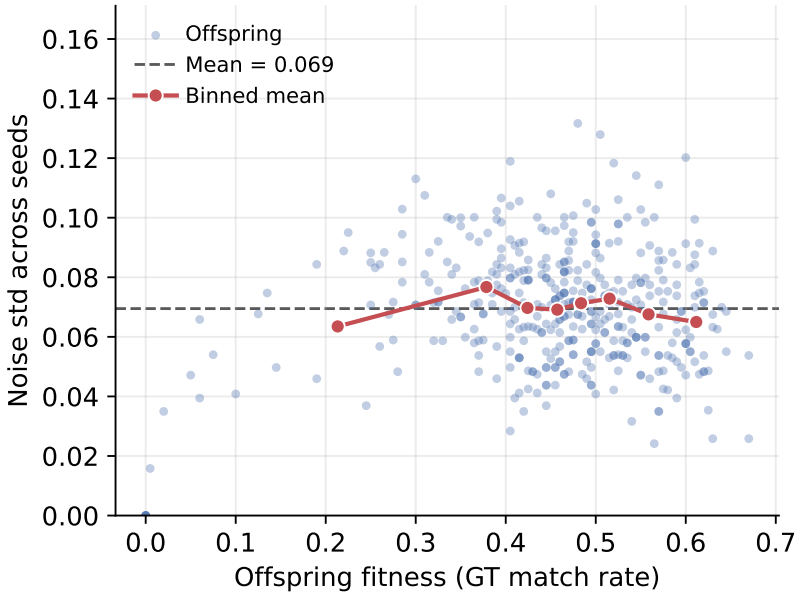


Figure 12: Evaluation noise is roughly constant across offspring fitness. Each point is one offspring from three independent $N_{init} = 10$ evolution runs (422 offspring total). Fitness is the GT match rate averaged over 20 training tasks and 10 PySR seeds. Noise std is the standard deviation across the 10 seeds of the per-seed fitness (averaged over tasks), i.e. the noise in a single-seed fitness estimate. The dashed line is the mean over all offspring (0.069). The red line is the mean within 8 equal-count quantile bins of fitness, which varies only from 0.064 to 0.077.

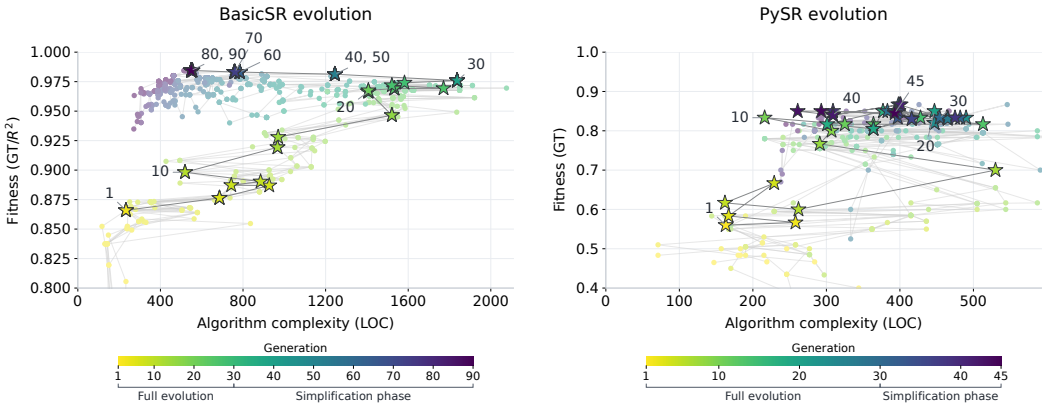


Figure 13: Complexity and fitness of the evolved population of algorithms for BasicSR (left) and PySR (right) evolution over time. Complexity is measured in lines of code of the combined set of evolved operators. Fitness is GT recovery for evolved PySR and combined ground truth recovery and R^2 for BasicSR. BasicSR evolved with 30 generations of full evolution and 60 generations of simplification; PySR evolved with 30 generations of full evolution and 15 generations of simplification. The best performing algorithm at each generation is shown with a star; the remaining population members are shown with a dot.

E Evolved algorithm interpretation

Figures 14 and 15 give a full description of the baseline and evolved operators for PySR and BasicSR. Figure 16 compares SRBench performance of the base PySR algorithm, the evolved PySR algorithm, and base PySR algorithm with one of the four operators selectively included. Loss and mutation improve performance on their own, but all four operators synergize to produce the best results.

Simplification. We find simplification phase crucial for interpreting the evolved operators. In particular, the algorithms evolved by BasicSR are extremely complex without simplification. We conducted 60 generations of simplification to BasicSR until the evolved algorithms plateaued in complexity. See Figure 13.

F Baseline and evolved operator sets

Below we show the baseline PySR and evolved operators: mutation, selection, survival, and loss functions.

F.1 PySR default operator set

The selection, survival, and loss function are PySR defaults. The default mutation operator in PySR is a weighted ensemble of seven mutation types; we create an additional custom mutation which is included in the ensemble with probability of roughly 5% of being chosen for any given mutation (the exact percentage depends on how many of the other mutation types are valid for a given expression.)

Listing 1: PySR baseline operators.

```
# Baseline operator set: add_constant_offset | age_regularized_survival |
    tournament_selection | mse_loss

""" Mutation: Wrap a random subtree with addition of a random constant. """
function add_constant_offset(
    tree::N,
    dataset,
    options,
    nfeatures::Int,
```

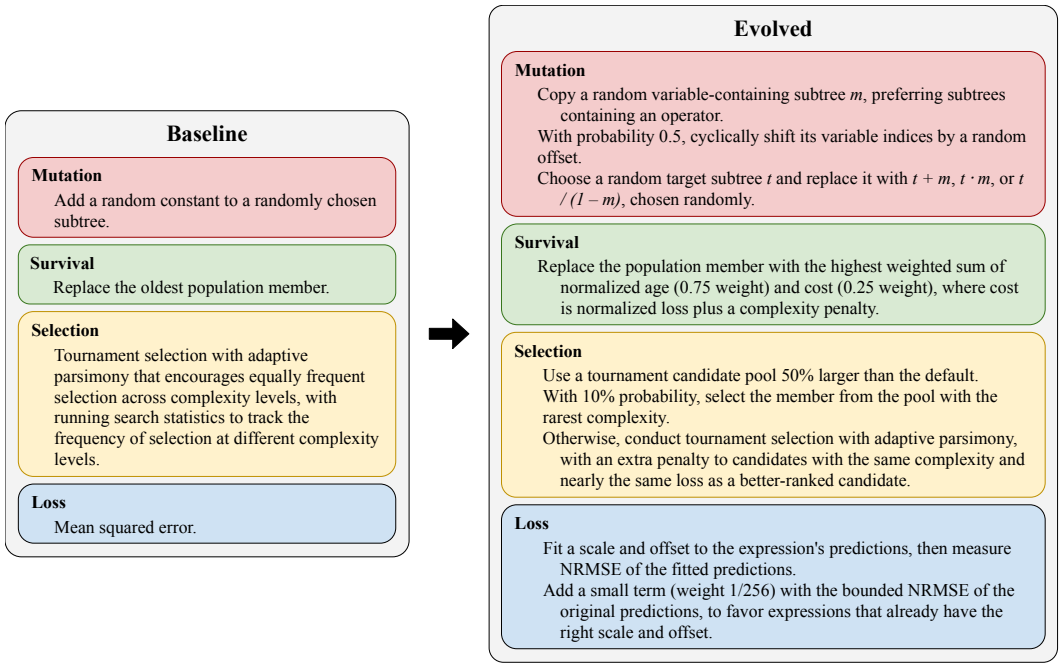


Figure 14: Operator sets for the PySR baseline and the evolved algorithm under the GT objective.

```
    rng::AbstractRNG,
) where {T,N<:AbstractExpressionNode{T}}
    # Find the + operator index (binary operators are indexed by their position)
    plus_idx = findfirst(op -> op == (+), options.operators.binops)

    if plus_idx === nothing
        # No + operator available, return tree unchanged
        return tree
    end

    # Sample a random node to wrap
    node = rand(rng, NodeSampler(; tree))

    # Create a random constant
    constant_value = randn(rng, T)  # Sample from normal distribution
    constant_node = constructorof(N)(T; val=constant_value)

    # Create new node: node + constant
    # Randomly decide if constant goes on left or right
    if rand(rng, Bool)
        new_node = constructorof(N)(; op=plus_idx, children=(copy(node), constant_node))
    else
        new_node = constructorof(N)(; op=plus_idx, children=(constant_node, copy(node)))
    end

    # Replace the selected node with the wrapped version
    set_node!(node, new_node)

    return tree
end

""" Default survival strategy: replace the oldest population member. """
function age_regularized_survival(
    pop::Population{T,L,N},
    options::AbstractOptions;
    exclude_indices::Vector{Int}=Int[],
)::Int where {T<:DATA_TYPE,L<:LOSS_TYPE,N}
    BT = typeof(first(pop.members).birth)
    births = [(i in exclude_indices) ? typemax(BT) : pop.members[i].birth
              for i in 1:(pop.n)]
    return argmin_fast(births)
end
```

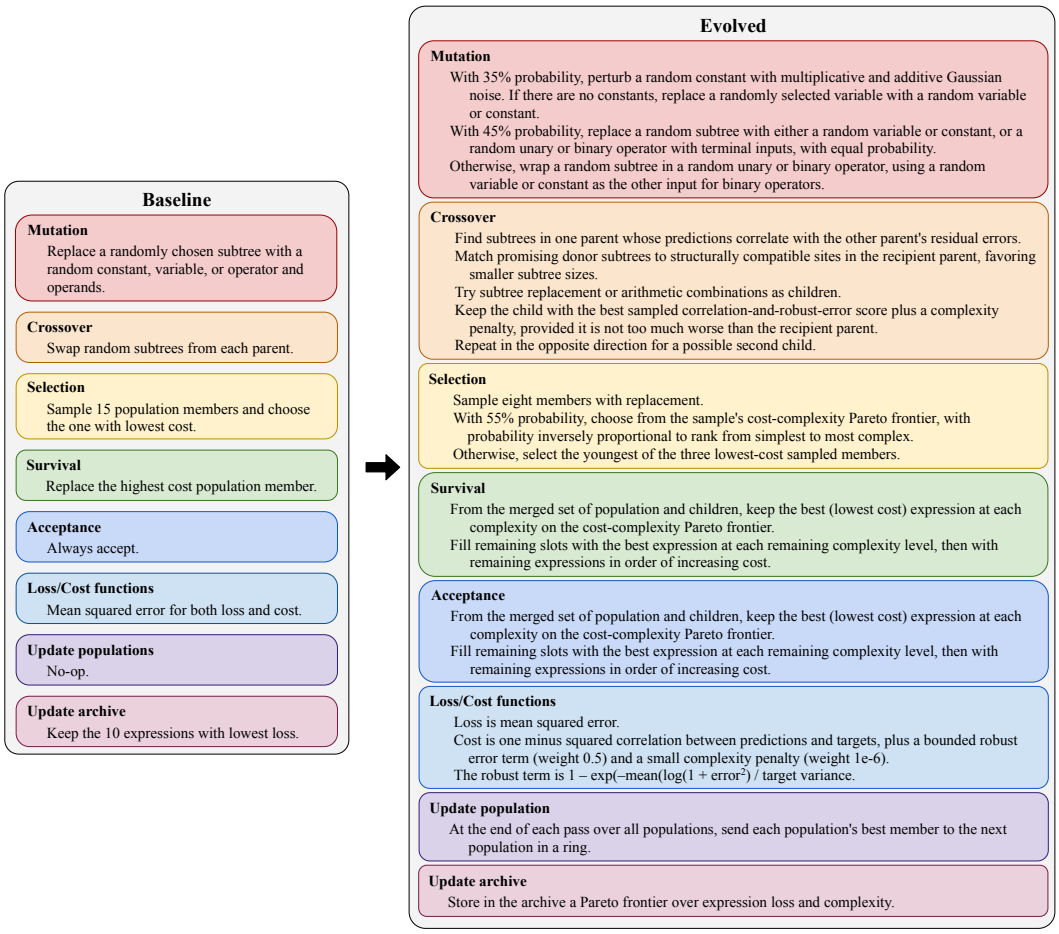


Figure 15: Operator sets for the BasicSR baseline and the evolved algorithm under the GT-R^2 objective.

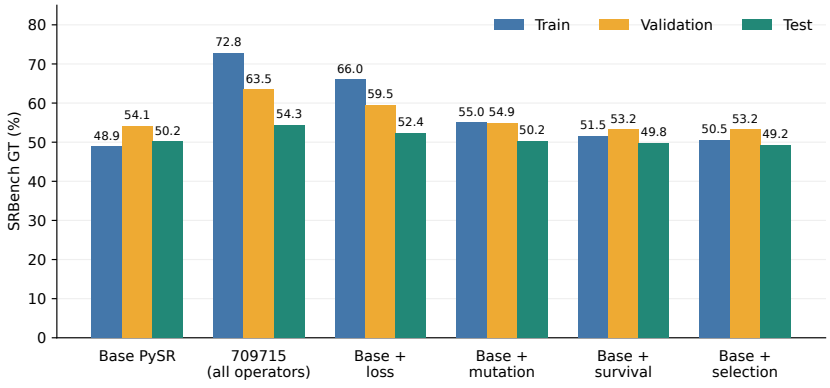


Figure 16: Operator ablations: evaluating base PySR with one of the four evolved operators from the ground truth evolution run.

```julia
""" Default selection strategy: tournament selection with adaptive parsimony. """
function tournament_selection(
    pop::Population{T,L,N},
    running_search_statistics::RunningSearchStatistics,
    options::AbstractOptions,
)::PopMember{T,L,N} where {T<:DATA_TYPE,L<:LOSS_TYPE,N}
    sample = StatsBase.sample(pop.members, options.tournament_selection_n; replace=false)
    n = length(sample)
    p = options.tournament_selection_p

    adjusted_costs = Vector{L}(undef, n)
    if options.use_frequency_in_tournament
        adaptive_parsimony_scaling = L(options.adaptive_parsimony_scaling)
        for i in 1:n
            member = sample[i]
            size = compute_complexity(member, options)
            frequency = if (0 < size <= options.maxsize)
                L(running_search_statistics.normalized_frequencies[size])
            else
                L(0)
            end
            adjusted_costs[i] = member.cost * exp(adaptive_parsimony_scaling * frequency)
        end
    else
        for i in 1:n
            adjusted_costs[i] = sample[i].cost
        end
    end

    chosen_idx = if p == 1.0
        argmin_fast(adjusted_costs)
    else
        k = collect(0:(n - 1))
        prob_each = p * ((1 - p) .^ k)
        weights = StatsBase.Weights(prob_each, sum(prob_each))
        tournament_winner = StatsBase.sample(weights)
        if tournament_winner == 1
            argmin_fast(adjusted_costs)
        else
            bottomk_fast(adjusted_costs, tournament_winner)[2][end]
        end
    end
    return copy(sample[chosen_idx])
end

""" Default loss: per-sample MSE. """
function mse_loss(
    tree::Union{AbstractExpression{T},AbstractExpressionNode{T}},
    dataset::Dataset{T,L},
    options::AbstractOptions,
)::L where {T<:DATA_TYPE,L<:LOSS_TYPE}
    prediction, completed = eval_tree_array(tree, dataset.X, options)
    if !completed || isnothing(prediction)
        return L(Inf)
    end
    diff = prediction .- dataset.y
    return L(sum(abs2, diff) / length(diff))
end
```

F.2 EVOLVED OPERATOR SET

The evolved operator set under the ground truth objective is shown below. Comments have been edited for brevity and clarity and the function implementations have been lightly edited to reduce the LOC while keeping equivalent functionality.

Listing 2: Evolved PySR operators.

```julia
# Evolved operator set
# Operators: motif_duplication_simple_rational_gen27_9 | age_and_cost_regularized_survival_simple_gen28_8 | streamlined_niche_clone_tournament_gen43_3 | simplified_affine_profile_loss_gen34_7

# === mutation: motif_duplication_simple_rational_gen27_9 ===
"""
Simplified structural mutation that **reuses evolved structure**: it clones a subexpression *motif*,
optionally shifts its feature indices, and couples the clone back into the tree either as a plain
accumulation (`target + motif` / `target * motif`) or inside a **self-referential rational template**
(`target / (1 - motif)`).

# Steps
1. Guard: need at least one binary operator; locate `+`, `-`, `*`, `/` once.
2. Decide the coupling template first, since the two cost a different number of extra nodes
   (plain: `size(motif)+1`; rational: `size(motif)+3`), and derive the donor size budget from
   `options.maxsize` so the child stays admissible.
3. Sample a donor motif: prefer *compound* variable-bearing subtrees that fit the budget, else any
   variable-bearing subtree that fits.
4. With probability 1/2 apply a cyclic feature shift to the clone (replicating the form across a
   block of variables); otherwise keep it identical (giving squares `u*u` and self-couplings
   `u/(1-u)`).
5. Splice: `target / (1 - motif)` for the rational template, otherwise `target ⊕ motif` with
   `⊕ ∈ {+, *}`.
"""
function motif_duplication_simple_rational_gen27_9(
    tree::N,
    options,
    nfeatures::Int,
    rng::AbstractRNG,
) where {T,N<:AbstractExpressionNode{T}}
    # ------------------------------------------------------------------ Step 1: guard + operator lookup
    options.nops[2] == 0 && return tree
    binops = options.operators.binops; nbin = options.nops[2]
    _op(f) = findfirst(i -> binops[i] === f, 1:nbin)
    idx_add = _op(+); idx_sub = _op(-); idx_mul = _op(*); idx_div = _op(/)

    # Canonical accumulators for the plain coupling mode
    accum = Int[]
    idx_add !== nothing && push!(accum, idx_add)
    idx_mul !== nothing && push!(accum, idx_mul)

    # ------------------------------------------------------------------ Step 2: template + size budget
    tree_size = count_nodes(tree)
    maxsize = hasproperty(options, :maxsize) ? options.maxsize : typemax(Int)

    # `1 - u` is the canonical correction form; fall back to `1 + u` if `-` is unavailable
    den_op = idx_sub !== nothing ? idx_sub : idx_add
    rational_ok = (idx_div !== nothing) && (den_op !== nothing) && (maxsize - tree_size - 3) >= 1

    # Rational is the minority mode (35%), unless it is the only feasible coupling
    use_rational = rational_ok && (isempty(accum) || rand(rng) < 0.35)
    (!use_rational && isempty(accum)) && return tree

    budget = use_rational ? (maxsize - tree_size - 3) : (maxsize - tree_size - 1)
    budget < 1 && return tree

```

```
    # ------------------------------------------------------------ Step 3: donor motif (two tiers)
    # Candidate motifs must fit the budget and carry at least one variable
    fits_var(t) =
        count_nodes(t) <= budget && any(n -> n.degree == 0 && !n.constant, t)

    donor = if count(t -> t.degree > 0 && fits_var(t), tree) > 0
        # Compound motifs are the meaningful symmetry / structure carriers
        rand(rng, NodeSampler(; tree, filter=t -> t.degree > 0 && fits_var(t)))
    elseif count(fits_var, tree) > 0
        rand(rng, NodeSampler(; tree, filter=fits_var))
    else
        return tree  # nothing usable fits the budget
    end
    motif = copy(donor)  # copy before any in-place edits to the tree

    target = rand(rng, NodeSampler(; tree))

    # ------------------------------------------------------------ Step 4: feature remapping
    # Single compact traversal that shifts every variable leaf index
    function _shift_features!(node, shift)
        if node.degree == 0
            if !node.constant; node.feature = mod1(node.feature + shift, nfeatures); end
        else
            for i in 1:(node.degree)
                _shift_features!(get_child(node, i), shift)
            end
        end
        return node
    end

    # Half the time replicate the form on a different variable block; otherwise keep it identical
    # (identity yields squares `u*u` and self-couplings `u/(1-u)`).
    if nfeatures > 1 && rand(rng) < 0.5; _shift_features!(motif, rand(rng, 1:(nfeatures - 1))); end

    # ------------------------------------------------------------ Step 5: coupling
    target_copy = copy(target)
    new_node = if use_rational
        # Self-referential saturation / correction factor: target / (1 - motif)
        c = constructorof(N)(T; val=one(T))
        den = constructorof(N)(; op=den_op, children=(c, motif))
        constructorof(N)(; op=idx_div, children=(target_copy, den))
    else
        # Plain accumulation with a commutative operator, so operand order does not matter
        bin_op = accum[rand(rng, 1:length(accum))]
        constructorof(N)(; op=bin_op, children=(target_copy, motif))
    end

    set_node!(target, new_node)
    return tree
end

# === survival: age_and_cost_regularized_survival_simple_gen28_8 ===
"""
Simplified age-with-cost-tiebreak survival: combines normalized age and
normalized cost directly into a single score instead of computing two
separate rank permutations.

Steps:
1. Collect eligible members' birth and cost values.
2. Min-max normalize births so oldest (smallest birth) -> 1.0, and normalize
   costs so worst (largest cost) -> 1.0.
3. Combine with age dominant weight (0.75) and cost as tie-break (0.25).
4. Return eligible index with highest combined score.
"""
function age_and_cost_regularized_survival_simple_gen28_8(
    pop::Population{T,L,N},
    options::AbstractOptions;
    exclude_indices::Vector{Int}=Int[],
)::Int where {T<:DATA_TYPE,L<:LOSS_TYPE,N}
    n = pop.n
    eligible = [i for i in 1:n if !(i in exclude_indices)]
    @assert !isempty(eligible) "No eligible members to replace"

    # Extract raw birth and cost values for eligible members
    births = [pop.members[i].birth for i in eligible]
    costs = Float64[pop.members[i].cost for i in eligible]
```

```

    # Min-max normalize age so that oldest (smallest birth) -> 1.0
    min_b, max_b = minimum(births), maximum(births)
    age_score = if max_b == min_b
        ones(Float64, length(eligible))  # all same age -> tie, defer to cost
    else
        [(max_b - b) / (max_b - min_b) for b in births]
    end

    # Min-max normalize cost so that worst (largest cost) -> 1.0
    min_c, max_c = minimum(costs), maximum(costs)
    cost_score = if max_c == min_c
        zeros(Float64, length(eligible))  # all same cost -> no tie-break signal
    else
        [(c - min_c) / (max_c - min_c) for c in costs]
    end

    # Combine: age dominates (0.75), cost nudges ties toward removing worse members
    age_weight = 0.75
    combined_score = age_weight .* age_score .+ (1 - age_weight) .* cost_score

    # Pick eligible member with highest combined score (oldest & worst-biased)
    best_local = argmax(combined_score)
    return eligible[best_local]
end

# === selection: streamlined_niche_clone_tournament_gen43_3 ===
"""
Streamlined niche-rescue and clone-penalizing tournament selection.

Steps:
1. Direct Niche Rescue: occasionally pick the best member of the rarest complexity in the pool
2. Clone Suppression: penalize members that duplicate a better pool member's complexity and loss
3. Standard Tournament Ladder over diversity-adjusted scores
"""
function streamlined_niche_clone_tournament_gen43_3(
    pop::Population{T,L,N},
    running_search_statistics::RunningSearchStatistics,
    options::AbstractOptions,
)::PopMember{T,L,N} where {T<:DATA_TYPE,L<:LOSS_TYPE,N}
    # Mild tunables for pool scaling, clone penalty factor, and niche rescue rate
    pool_scale = 1.5; clone_penalty = L(1.5); clone_loss_rtol = L(1e-6); niche_probability = 0.10

    # Sample an enlarged tournament pool to observe local diversity
    n_base = options.tournament_selection_n
    pool_size = min(pop.n, max(n_base, round(Int, pool_scale * n_base)))
    pool = StatsBase.sample(pop.members, pool_size; replace=false)
    n = length(pool)

    # Compute complexity and adaptive-parsimony-adjusted baseline scores
    complexities = Vector{Int}(undef, n); scores = Vector{L}(undef, n)
    adaptive_parsimony_scaling = L(options.adaptive_parsimony_scaling)

    for i in 1:n
        member = pool[i]; complexity = compute_complexity(member, options); complexities[i] = complexity

        if options.use_frequency_in_tournament
            frequency = if 0 < complexity <= options.maxsize
                L(running_search_statistics.normalized_frequencies[complexity])
            else
                zero(L)
            end
            scores[i] = member.cost * exp(adaptive_parsimony_scaling * frequency)
        else
            scores[i] = member.cost
        end
    end

    # 1. Direct Niche Rescue: occasionally pick the best member of the rarest complexity in the pool
    if rand() < niche_probability
        c1 = complexities[1]
        min_freq = (0 < c1 <= options.maxsize) ? Float64(running_search_statistics.normalized_frequencies[c1]) : 1.0
        best_score = scores[1]; best_idx = 1

        for i in 2:n
```

```
            c = complexities[i]
            freq = (0 < c <= options.maxsize) ? Float64(running_search_statistics.normalized_frequencies[c]) : 1.0
            # Prefer rarer complexity niches; break ties with lower score
            if freq < min_freq || (freq == min_freq && scores[i] < best_score)
                min_freq = freq; best_score = scores[i]; best_idx = i
            end
        end

        return pool[best_idx]
    end

    # 2. Clone Suppression: penalize members that duplicate a better pool member's complexity and loss
    order = sortperm(scores); adjusted_scores = copy(scores)

    for i in 2:n
        idx_i = order[i]; loss_i = pool[idx_i].loss; scale_i = max(abs(loss_i), eps(L))

        for j in 1:(i - 1)
            idx_j = order[j]; loss_j = pool[idx_j].loss; loss_scale = max(scale_i, abs(loss_j))

            # Near-identical loss at identical complexity marks a redundant clone
            if complexities[idx_i] == complexities[idx_j] && abs(loss_i - loss_j) <= clone_loss_rtol * loss_scale
                adjusted_scores[idx_i] = if adjusted_scores[idx_i] > zero(L)
                    adjusted_scores[idx_i] * clone_penalty
                elseif adjusted_scores[idx_i] < zero(L)
                    adjusted_scores[idx_i] / clone_penalty
                else
                    adjusted_scores[idx_i] + L(0.1)
                end
                break
            end
        end
    end

    # 3. Standard Tournament Ladder over diversity-adjusted scores
    p = options.tournament_selection_p
    chosen_idx = if p == 1.0
        argmin_fast(adjusted_scores)
    else
        ranks = collect(0:(n - 1))
        probabilities = p * ((1 - p) .^ ranks)
        weights = StatsBase.Weights(probabilities, sum(probabilities))
        selected_rank = StatsBase.sample(weights)

        if selected_rank == 1
            argmin_fast(adjusted_scores)
        else
            bottomk_fast(adjusted_scores, selected_rank)[2][end]
        end
    end

    return pool[chosen_idx]
end

# === loss: simplified_affine_profile_loss_gen34_7 ===
"""
Score predictions by the residual of their best affine calibration, preserving the
parent operator's preference for expressions with the correct functional shape.
The loss computes sample means, fits a least-squares slope from centered moments,
and directly evaluates the calibrated residual. A small bounded raw-NRMSE term
then favors expressions whose constants are already correctly calibrated.

Steps:
1. First pass: online means avoid overflow-prone raw sums.
2. Second pass: centered moments for the affine least-squares slope.
3. Third pass: direct residual evaluation remains accurate near an exact fit.
"""
function simplified_affine_profile_loss_gen34_7(
    tree::Union{AbstractExpression{T},AbstractExpressionNode{T}},
    dataset::Dataset{T,L},
    options::AbstractOptions,
)::L where {T<:DATA_TYPE,L<:LOSS_TYPE}
    prediction, completed = eval_tree_array(tree, dataset.X, options)
    if !completed || isnothing(prediction); return L(Inf); end

    n = dataset.n
    if n <= 0 || length(prediction) != n || length(dataset.y) != n; return L(Inf); end
```

```

    zero_l = zero(L); one_l = one(L); n_l = L(n)

    # First pass: online means avoid overflow-prone raw sums.
    mean_p = zero_l; mean_y = zero_l

    @inbounds for i in 1:n
        p = L(prediction[i]); y = L(dataset.y[i])
        if !(isfinite(p) && isfinite(y)); return L(Inf); end

        k = L(i)
        mean_p += (p - mean_p) / k; mean_y += (y - mean_y) / k
    end

    if !(isfinite(mean_p) && isfinite(mean_y)); return L(Inf); end

    # Second pass: centered moments for the affine least-squares slope.
    m2_p = zero_l; m2_y = zero_l; cov_py = zero_l

    @inbounds for i in 1:n
        dp = L(prediction[i]) - mean_p; dy = L(dataset.y[i]) - mean_y
        m2_p += dp * dp; m2_y += dy * dy; cov_py += dp * dy
    end

    if !(isfinite(m2_p) && isfinite(m2_y) && isfinite(cov_py)); return L(Inf); end

    # A small scale-relative floor handles constant and nearly constant targets
    # without a separate loss branch.
    target_scale = max(one_l, abs(mean_y))
    target_floor = sqrt(eps(L)) * sqrt(n_l) * target_scale
    target_norm = max(sqrt(max(m2_y, zero_l)), target_floor)
    if !(isfinite(target_norm) && target_norm > zero_l); return L(Inf); end

    # Exactly constant predictions cannot supply a meaningful affine direction.
    slope = m2_p > zero_l ? cov_py / m2_p : zero_l
    if !isfinite(slope); return L(Inf); end

    # Third pass: direct residual evaluation remains accurate near an exact fit.
    affine_sum = zero_l; raw_sum = zero_l

    @inbounds for i in 1:n
        p = L(prediction[i]); y = L(dataset.y[i])

        fitted = mean_y + slope * (p - mean_p)
        affine_residual = (fitted - y) / target_norm
        affine_term = affine_residual * affine_residual

        if !(isfinite(fitted) && isfinite(affine_term)); return L(Inf); end
        affine_sum += affine_term
        if !isfinite(affine_sum); return L(Inf); end

        raw_residual = p - y
        if !isfinite(raw_residual); return L(Inf); end

        # Raw overflow is harmless because this component is intentionally bounded.
        raw_normalized = raw_residual / target_norm
        raw_sum += raw_normalized * raw_normalized
    end

    shape_term = sqrt(clamp(affine_sum, zero_l, one_l))

    bounded_raw = if isfinite(raw_sum)
        raw_nrmse = sqrt(max(raw_sum, zero_l))
        raw_nrmse / (one_l + raw_nrmse)
    else
        one_l
    end

    loss_value = shape_term + bounded_raw / L(256)
    return isfinite(loss_value) ? L(max(loss_value, zero_l)) : L(Inf)
end
```