EliteTable: Self-Improving Program Search for Executable Control Policies

Stuart Farmer

Casa Dilauro Independent Research (casadilauro.com)

stuart@casadilauro.com

github.com/StuartFarmer/elitetable

Abstract—EliteTable is a self-improving program-search algorithm guided by large language models (LLMs) that uses a fixed-capacity table of high-scoring solutions to guide new proposals, edits of existing solutions, and remixes of several solutions. Performance was evaluated across six Gymnasium control tasks, each with ten generations and fifty candidate slots per generation. Each task was evaluated across ten independently initialized searches. Candidate-score medians rose between the first and final generations in all six tasks, and selected-winner search-score medians rose in five. In `CartPole-v1`, the selected winner reached the search-score ceiling in the first generation of all ten searches. All ten paired held-out endpoints improved in four of the six tasks. `BipedalWalker-v3` and `CarRacing-v3` improved without reaching their reporting targets. Six of ten `Acrobot-v1` searches declined on held-out episodes, and the median paired change was negative. The results document improvement during program search and the transfer of selected-winner gains to new episodes in some tasks. Comparisons and ablations are needed to establish EliteTable's relative performance and the contributions of its components.

1 Introduction

This work was motivated by the broader study of recursive self-improvement and LLM-guided program search: how these algorithms are structured, which tasks they address, how well their solutions generalize, and which mechanisms support improvement.

The study examines one part of that motivation: whether a program-search algorithm inspired by prior work can also show improvement through repeated evaluation and revision.

In program search, one part of the improvement loop proposes a solution, often source code or an executable policy. Another part, the evaluator, provides a score or feedback on how that solution performs on the task.

Optimization by PROmpting (OPRO) illustrates this feedback pattern: a language model receives a task description and previously evaluated solutions, then uses that feedback to propose further solutions [1].

FunSearch makes this feedback loop concrete for executable programs. It uses an island-based database of solutions, maintaining multiple populations to support diversity. Its applications include cap-set constructions and online bin-packing heuristics [2].

AlphaEvolve extends evolutionary code modification to scientific and computational problems, using evaluator feedback to guide revisions [3]. In these systems, evaluation gives program search an external measure of whether a proposed change is useful.

These algorithms differ in what they retain and how they generate candidates. *ShinkaEvolve* combines parent sampling, code-novelty rejection, and bandit-based selection among language models [4].

Evolution of Heuristics evolves descriptions of heuristic ideas together with their executable implementations, giving the language model both conceptual and code-level information about the solutions it explores [5].

EliteTable draws on these approaches through a fixed-capacity table of high-scoring programs. New candidates compete with retained solutions after each generation, while three operators generate fresh proposals, edits, and remixes. The design aims to provide a simple feedback loop for repeated program improvement.

Keeping incumbents eligible guarantees that the best recorded search score cannot decrease, but does not guarantee that newly generated solutions improve. A search may retain elites based on similar heuristics and become confined to a local optimum [2]. Selected solutions may also fail to generalize to new episodes, so held-out evaluation is necessary [10].

The study asks two questions. First, do in-sample winner scores and candidate-score distributions improve across generations? Second, do selected-winner gains transfer to held-out episodes? I examine these questions across six Gymnasium control tasks [7].

Records were kept for every candidate across the sixty searches. Between generations one and ten, pooled candidate-score medians increased in all six tasks and median selected-winner search scores increased in five. `CartPole-v1` winners were already at the ceiling in generation one.

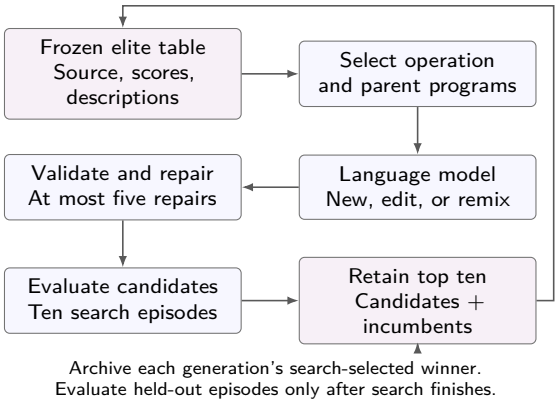

Figure 1: EliteTable's evaluation–revision loop. All candidate slots finish before the table is updated. Held-out scores do not enter the loop.

In `CarRacing-v3`, the candidate population showed a bimodal score distribution: a higher-return concentration moved toward stronger scores across generations while a low-return group persisted.

Held-out evaluation addressed the second question. All ten paired endpoint changes were positive in four tasks. `Acrobot-v1`'s median paired held-out change was negative. `BipedalWalker-v3` and `CarRacing-v3` improved between the first and final evaluations, but neither reached its reporting target.

Held-out gains were not necessarily monotonic: intermediate performance sometimes declined. Longer searches might produce further gains, but the tested budget does not establish whether `BipedalWalker-v3` or `CarRacing-v3` would eventually reach their targets.

2 EliteTable Program Search

Figure 1 shows the search loop, summarized in *Algorithm 1*. The retained table supplies context for subsequent proposals and preserves previously discovered high-scoring programs.

2.1 Generation and repair

EliteTable uses three operators: new, edit, and remix. Generation one contains only fresh proposals; the top K evaluated solutions enter the elite table. Subsequent generations use the allocation in *Table 2*. Fresh proposals receive elite descriptions and scores, edits receive one parent's source, and remixes combine information from up to three parents. Parents are sampled uniformly from the table frozen at the start of the generation.

Before evaluation, proposed solutions are checked for schema compliance, valid source edits, and parseable Python syntax using the abstract syntax tree (AST).

Solutions must satisfy the required policy interface and pass duplicate checks. Duplicate detection compares AST structures rather than source text.

Solutions are then passed to the evaluator. Validation or execution failures, such as access to an undefined variable, return diagnostics to a repair loop so the model can revise the proposal.

If the revised solution still fails, it returns to the repair loop, up to five repairs per candidate slot.

Candidate timeouts are treated as failures, not zero-reward episodes or successful truncations. Infrastructure failures interrupt execution rather than enter the candidate-repair loop.

Algorithm 1 EliteTable search

Require: Task, fixed LLM, search panel; $T = 10$, $N = 50$, $K = 10$
$\mathcal{E}_0 \leftarrow \emptyset$
for $t = 1, \ldots, T$ **do**
  Freeze parent table at $\mathcal{E}_{t-1}$
  Allocate new/edit/remix slots (*Table 2*)
  Apply empty/single-elite fallbacks; shuffle slots
  Sample parents uniformly; remix parents are distinct
  $\mathcal{C}_t \leftarrow \emptyset$
  for all candidate slots **do**
    Generate program; validate, evaluate, and repair
    Add successful evaluated program to $\mathcal{C}_t$
  end for
  Wait for every slot to finish or be discarded
  $\mathcal{E}_t \leftarrow \mathrm{TopK}(\mathcal{E}_{t-1} \cup \mathcal{C}_t)$
  Archive the search-selected winner in $\mathcal{E}_t$
end for
Evaluate archived winners on held-out episodes

2.2 Elite selection

As briefly mentioned, EliteTable retains the highest-scoring programs across completed generations. For elite capacity K, the update rule is

$$\mathcal{E}_t = \mathrm{TopK}\left(\mathcal{E}_{t-1} \cup \mathcal{C}_t\right). \tag{1}$$

Here $\mathcal{E}_t$ is the retained elite set after generation t, $\mathcal{C}_t$ contains valid new candidates from that generation, and TopK selects up to K programs. Candidates are ranked by descending mean search reward.

For equal scores, edits take priority over remixes, followed by incumbent elites and then fresh proposals. Any remaining ties are resolved by ascending organism identifier.

This persistent elitist mechanism ensures that the highest recorded *search-panel* reward cannot decrease between completed generations, assuming previously recorded evaluations remain unchanged. This monotonicity does not imply monotonic held-out performance.

3 Experimental Design

3.1 Study structure

Six control environments were selected from the Gymnasium reinforcement learning library [7]. Their task profiles are listed in *Table 1*.

Environment	Observation/action interface	Step cap	Target	Task Settings
`CartPole-v1`	State vector; discrete	500	475	Recorded environment profile.
`MountainCar-v0`	State vector; discrete	200	−110	Recorded environment profile.
`Acrobot-v1`	State vector; discrete	500	−100	Recorded environment profile.
`LunarLander-v3`	State vector; discrete	1000	200	Gravity −10; wind disabled.
`BipedalWalker-v3`	State vector; continuous	1600	300	Standard, non-hardcore task.
`CarRacing-v3`	96 × 96 RGB; continuous	1000	900	Continuous control; lap-completion fraction 0.95; domain randomization disabled. Policies derive control directly from native RGB observations, so this task is interpreted separately from state-vector tasks.

Table 1: Task suite, settings and descriptive reporting targets. Other constructor settings follow the recorded environment profiles. RGB denotes red, green, and blue image channels.

The tasks were selected for their perceived differences in difficulty and their variety of observation and action spaces.

Across the sixty searches, hyperparameters were chosen to balance computational cost against the opportunity for improvement.

Hyperparameter	**Setting**
Tasks	6
Searches per task	10
Population size	50
Elite table size	10
Generations	10
Initial operator split	100% new, 0% edit, 0% remix
Subsequent operator split	20% new, 40% edit, 40% remix
Search initialization seeds	$\{0, \ldots, 9\}$
In-sample seeds	$\{0, \ldots, 9\}$
Out-of-sample seeds	$\{1000, \ldots, 1099\}$
Maximum repairs per slot	5
Early stopping?	False

The subsequent operator split applies when at least two elites are available. Held-out episodes are evaluated per distinct generation winner within each search.

Table 2: Shared search and evaluation settings.

Each candidate slot includes its initial proposal and any repair attempts; it either produces an evaluated policy or is discarded. Slot counts therefore differ from counts of valid programs, model requests, and completed episode evaluations.

The study records the configuration and execution history of each search. Initialization seeds control operator scheduling and parent sampling, but do not make hosted model responses deterministic. Reproduction can therefore follow the recorded procedure without guaranteeing identical search trajectories.

3.2 Environments and policy interface

Generated solutions must implement `Solution(Policy)`, a Python class with an asynchronous action method. Policy objects receive the current observation, observation and action spaces, and task instructions.

The policy interacts with the environment through an observe–act loop. Its runtime interface does not expose reward values or environment internals.

A policy may retain state within an episode, provided it satisfies the required interface. That state must be reset at the start of each episode.

Generation and repair prompts include task documentation, resolved environment settings, the observation/action contract, the objective, and execution constraints. This documented task knowledge is distinct from the information exposed to a policy during execution.

For `BipedalWalker-v3`, the prompts also include explicit observation indexing, normalization, motor ordering, and reward details. Prompt construction is configurable by task.

The study therefore evaluates program search with documented task knowledge, rather than learning solely from interaction.

3.3 Model configuration

I used the requested model route `openai/gpt-oss-120b:nitro` through OpenRouter for proposal generation and repair.

Responses were capped at 16,384 tokens per request.

The runner does not explicitly set temperature or top-p, so the study does not specify fixed values for those parameters. Returned model, provider, and usage metadata are retained when available. The requested route does not guarantee identical provider execution across requests.

Because fresh proposals receive elite summaries, they cannot serve as a history-independent sampling baseline.

3.4 Fitness and held-out evaluation

For policy π, its episode return under seed s is the undiscounted sum

$$R(\pi, s) = \sum_{t=0}^{T_s - 1} r_t, \tag{2}$$

where T_s is the number of steps before environment termination or the configured step cap.

Candidate fitness is the arithmetic mean over the ten fixed search seeds $S = \{0, \ldots, 9\}$:

$$F(\pi) = \frac{1}{10} \sum_{s \in S} R(\pi, s). \tag{3}$$

EliteTable ranks candidates by higher fitness scores. Tasks formulated as minimization problems would require an appropriate score transformation.

All candidate slots must finish evaluation or be discarded before the elite table is updated. The top K programs are then selected from the new evaluated candidates and existing elites, using the tie-breaking order described above.

After search finishes, the fixed historical winner from every generation is evaluated on 100 held-out episode seeds,

$$H = \{1000, \ldots, 1099\},$$

which are disjoint from both the search panel and the previously inspected pilot panel 100–199.

Each distinct selected policy is evaluated once per held-out seed within a search. Repeated generation winners within that search reuse cached measurements. Held-out outcomes do not influence generation, repair, promotion, stopping, or winner selection.

For environment e, search i, and generation g, held-out performance is

$$Y_{e,i,g} = \frac{1}{100} \sum_{s \in H} R(\pi^*_{e,i,g}, s), \tag{4}$$

where $\pi^*_{e,i,g}$ is the winner selected by search fitness.

A valid held-out mean requires finite returns on all 100 specified seeds and no recorded evaluation failures. Partial means are excluded. Missing evaluations remain missing; they are not replaced with zero or rerun.

The paired held-out change is $\Delta_{e,i} = Y_{e,i,10} - Y_{e,i,1}$. I report the median of these ten within-search changes for each task, rather than the difference between the first- and final-generation medians.

Descriptive 95% percentile intervals use 10,000 bootstrap resamples of the ten whole-search paired records, with replacement. Each resample preserves the within-search pairing and recomputes the median change. These intervals describe variation conditional on the fixed episode panels and tested configuration; they do not establish robustness across new panels or models.

As a retrospective exploratory analysis, I applied two-sided exact paired sign tests [11] to the ten endpoint changes per task. The null is $\Pr(\Delta > 0 \mid \Delta \neq 0) = 1/2$, not equality of mean returns. Exact zero changes are excluded from sign counts. Holm adjustment [12] covers all six tasks, retaining an all-tied task as $p = 1$ for correction only. These tests assume independent searches and condition on the fixed episode panels and tested configuration. They do not establish comparative superiority or causal operator effects.

3.5 Execution and reproducibility

Sixty searches were completed: ten initialization seeds across six tasks. Each began at generation zero with no existing solutions. Evaluation ran on an AMD64 host using eight worker processes, with up to 25 concurrent model operations for generation and repair. The configured deadlines were 45 seconds per episode, ten seconds per policy action, and 120 seconds per model-operation attempt.

The episode and action deadlines exclude Docker startup, queueing, and final cleanup. Generated policies ran in Docker containers with networking disabled, a read-only root filesystem, and resource limits.

Core reproduction code and recorded evidence are available at https://github.com/StuartFarmer/elitetable. The accompanying reproduction guide identifies the pinned revision and offline reconstruction commands. The retrospective sign-test extension and additional inputs and scripts for supplemental analyses accompany this manuscript locally; they are not included in that pinned public release. Reconstructing recorded results and running new stochastic searches are distinct procedures.

4 Experimental Results

4.1 Completion and improvement during search

All sixty searches completed. Held-out measurements were available for 599 of 600 generation-winner panels. The missing `MountainCar-v0` search-seed-8 panel was at generation six and did not affect any endpoint pair.

Median selected-winner search scores increased between the first and final generations in five environments. `CartPole-v1` winners were already at the score ceiling in generation one.

Newly evaluated candidate medians rose between generations one and ten in all six tasks, including `CartPole-v1`. *Figure 2* shows these population changes, extending the evidence beyond retention of strong initial winners.

These distributions contain newly evaluated candidates, not retained elites or held-out winners. Each score averages ten search episodes; failed or discarded candidates without scores are excluded. The `LunarLander-v3` display is restricted to returns between -1000 and 350: 198 scores outside this range remain in the calculations.

Candidate-score distributions show bimodality in some tasks. This motivated an exploratory analysis of whether fresh proposals, edits, and remixes contribute differently to the score mixture. The operator analyses appear in Supplemental Analyses.

4.2 Performance on held-out episodes

Table 3 reports paired endpoints and target counts. All ten searches improved in each of four environments: `MountainCar-v0`, `LunarLander-v3`, `BipedalWalker-v3`,

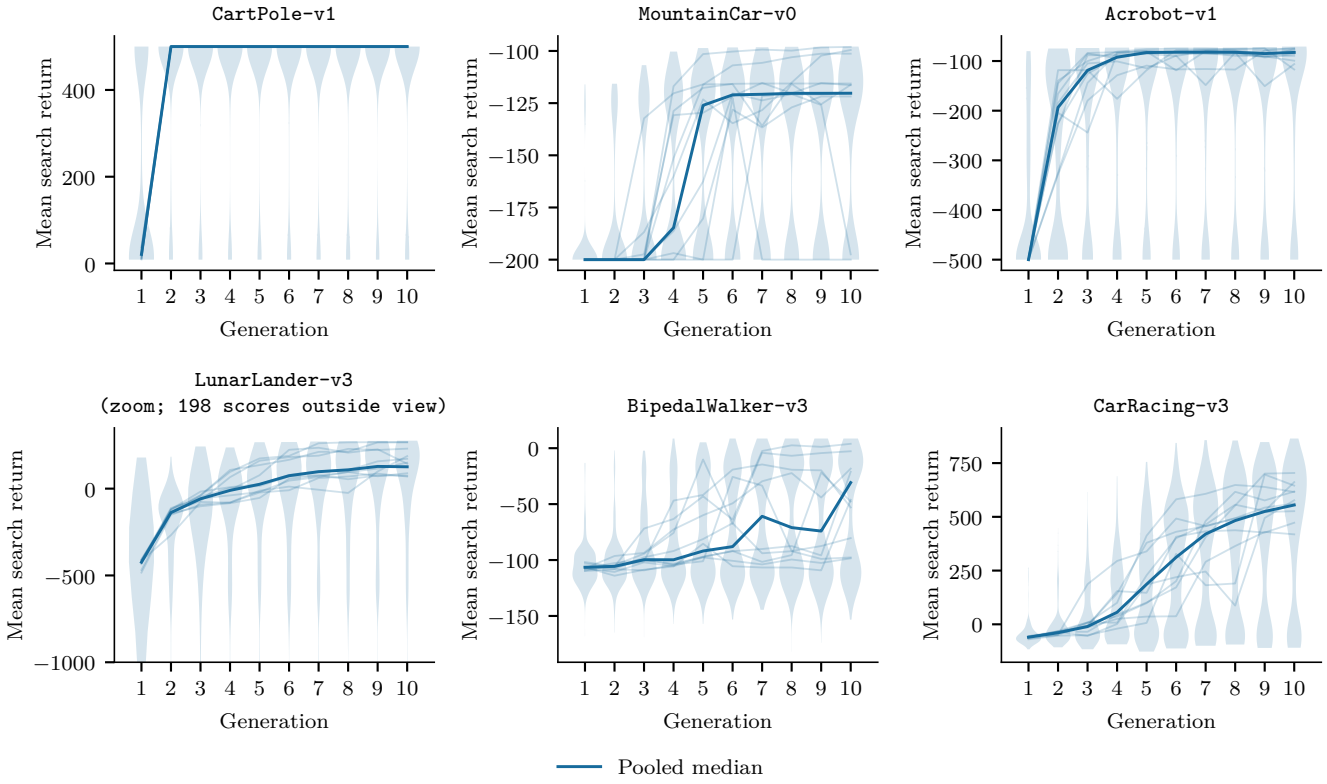

Figure 2: Candidate search-return distributions across generations, pooled over ten searches per task. Solid blue lines show pooled candidate medians; faint lines show medians within individual searches. Violin widths show smoothed density, not operator membership or independent replication.

Environment	First	Final	Change [95% CI]	Improved	Target	p_{Holm}
`CartPole-v1`	500.00	500.00	0.00 [0.00, 0.00]	0/10	10/10	—
`MountainCar-v0`	−119.18	−107.07	+7.41 [2.56, 22.47]	10/10	6/10	0.01172
`Acrobot-v1`	−94.83	−101.53	−6.38 [−18.64, 7.98]	4/10	3/10	1.00000
`LunarLander-v3`	−39.04	214.04	+189.31 [109.25, 260.20]	10/10	6/10	0.01172
`BipedalWalker-v3`	−39.30	−7.65	+42.27 [5.89, 55.94]	10/10	0/10	0.01172
`CarRacing-v3`	206.66	691.81	+507.11 [421.20, 629.13]	10/10	0/10	0.01172

Table 3: Held-out performance across ten searches per task. First and Final are medians of the generation-one and generation-ten mean returns. Change is the median paired change with a descriptive 95% bootstrap confidence interval (CI). Improved counts positive paired changes; Target counts final winners meeting the reporting threshold. Exploratory two-sided sign-test p-values are Holm-adjusted across all six tasks. A dash denotes ten ties and no informative test.

and `CarRacing-v3`. `CartPole-v1` remained at the ceiling, while six of ten `Acrobot-v1` searches declined. `BipedalWalker-v3` and `CarRacing-v3` improved without a final winner reaching the reporting target.

Each of the four tasks with ten positive changes has raw $p = 0.00195$ and Holm-adjusted $p = 0.01172$. For `Acrobot-v1`, raw $p = 0.75391$ and adjusted $p = 1$, so the observed negative median does not establish a systematic decline. The ten `CartPole-v1` ties provide no informative sign test and do not establish equivalence.

Figure 3 shows how outcomes develop across generations. Held-out scores sometimes decline even when the endpoint improves. `Acrobot-v1`'s positive search gains coexist with a negative median paired held-out change, consistent with possible selection-panel specialization. The study shows transfer of endpoint gains to new episodes in four tasks, but does not establish robustness across task families or rule out overfitting. Broader evaluation and comparisons with other search and reinforcement-learning methods are needed.

4.3 Candidate reliability and resource use

Of 30,000 candidate slots, 29,959 reached evaluated status (99.9%) and 41 were discarded. The logs contain 38,289 model calls, including 8,287 repair calls, or 21.6% of all calls. Repair calls amount to 27.6% of the 30,000 initial proposal calls; because a slot may undergo multiple

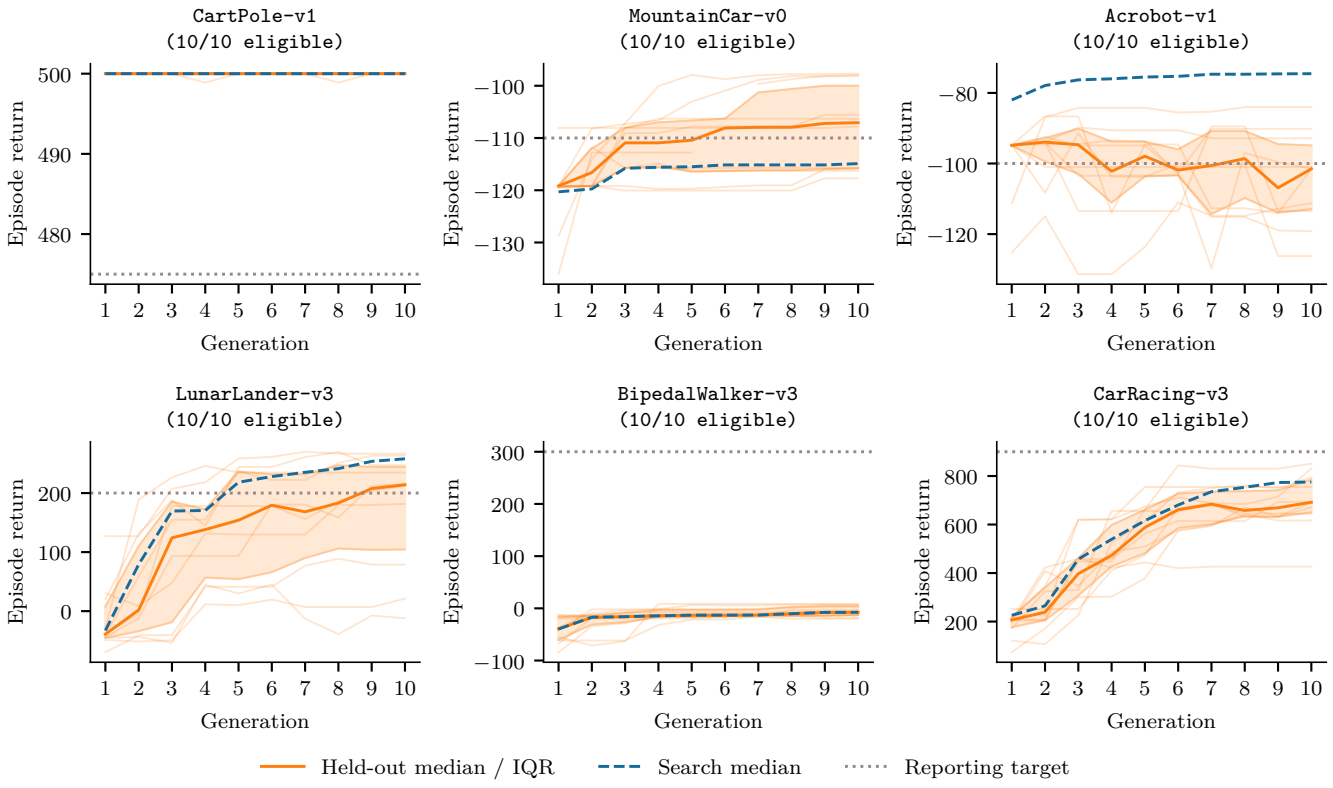

Figure 3: Search and held-out returns over ten generations. Dashed blue lines show median search return; solid orange lines show median held-out return, with interquartile ranges (IQRs) shaded and individual search trajectories drawn faintly. Dotted lines indicate the reporting threshold. Each environment contains ten eligible searches. `MountainCar-v0` generation 6 has nine valid held-out measurements; all other environment-generation combinations have ten. Shading represents between-search dispersion rather than a confidence interval.

repairs, this is not the percentage of proposals needing repair. The completion rate describes the combined procedure and does not isolate the repair loop's contribution.

5 Discussion and Conclusion

The rising candidate-score medians show progress beyond retaining a single strong initial solution. Newly generated populations also shift toward higher returns. In four tasks, every final selected winner improved over its first-generation counterpart on the fixed held-out episode panel. These endpoint gains do not by themselves rule out overfitting to the selection panel.

`Acrobot-v1` exposes a limitation: search scores rose while the median paired held-out change was negative. Further examination is needed to identify the cause. Held-out evaluation covers each generation's selected winner, not every retained elite, so it does not describe the full elite population. Search used ten episode seeds, compared with 100 held-out seeds. Future studies could test whether broader selection panels reduce specialization; the current design does not isolate that effect.

`BipedalWalker-v3` and `CarRacing-v3` improved without reaching their reporting targets. The tested budget does not establish whether additional generations would eventually solve these tasks. Their gains should therefore be distinguished from task attainment.

5.1 Program search and neural-policy training

EliteTable revises policy source code rather than updating neural-policy weights through policy gradients. Source edits can change numerical constants as well as program structure. Policy-gradient methods instead update parameterized policies, often neural networks, using estimates of the gradient of expected return with respect to policy parameters [8].

The selected tasks are drawn from Gymnasium's standard reinforcement-learning environments [7]. Matched comparisons are needed to assess the relative strengths of program search and neural-policy training on these tasks.

Source revisions can alter control structures, numerical constants, state, and execution flow. Programmatic reinforcement learning can also combine program search with guidance from a trained neural policy [9]. These are different update mechanisms; the present study provides no matched comparison of their learning efficiency.

5.2 Limitations and next steps

Understanding EliteTable's components requires ablations and comparisons with other systems. The observed improvements cannot yet be attributed specifically to reuse, editing, remixing, retention, or repair. Ten searches per environment quantify variation under fixed episode panels, but do not establish robustness across episode distributions, language models, prompts, or search configurations. The tasks are well known, so the model may have encountered descriptions or potential solutions during training.

Time-based deadlines can produce different outcomes on different hardware. Hosted-provider variation can also change model outputs and search trajectories, even when the same model route is requested.

The final solutions are Python programs available for inspection, but source availability alone does not establish human interpretability.

I would prioritize budget-matched comparisons with other program-search algorithms, including a history-independent generation baseline, followed by complete-search ablations and comparisons across language models. Longer budgets, new episode panels, and other task families would test the limits of the observations. The present results provide development evidence for those designs rather than confirmatory evidence of comparative superiority.

5.3 Conclusion

EliteTable contributes to the study of self-improving LLM-assisted program search. Across sixty searches, candidate populations improved on selection episodes, while held-out evaluation distinguished transferable gains, saturation, and regression. This work supports further investigation of repeated program improvement as part of the broader agenda toward recursive self-improvement. It does not test a system that modifies its own optimizer or language model.

References

[1] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large Language Models as Optimizers. *International Conference on Learning Representations*, 2024. Paper.

[2] Bernardino Romera-Paredes et al. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024. Article.

[3] Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. arXiv:2506.13131, 2025. Paper.

[4] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards Open-Ended And Sample-Efficient Program Evolution. arXiv:2509.19349, 2025. Paper.

[5] Fei Liu et al. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model. *Proceedings of the 41st International Conference on Machine Learning*, PMLR 235:32201–32223, 2024. Conference paper.

[6] Jacky Liang et al. Code as Policies: Language Model Programs for Embodied Control. arXiv:2209.07753, 2022; revised 2023. Paper.

[7] Mark Towers et al. Gymnasium: A Standard Interface for Reinforcement Learning Environments. arXiv:2407.17032, 2024; revised 2025. Paper.

[8] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy Gradient Methods for Reinforcement Learning with Function Approximation. *Advances in Neural Information Processing Systems 12*, 1999. Conference paper.

[9] Abhinav Verma, Vijayaraghavan Murali, Rishabh Singh, Pushmeet Kohli, and Swarat Chaudhuri. Programmatically Interpretable Reinforcement Learning. *Proceedings of the 35th International Conference on Machine Learning*, PMLR 80:5045–5054, 2018. Conference paper.

[10] Gavin C. Cawley and Nicola L. C. Talbot. On Overfitting in Model Selection and Subsequent Selection Bias in Performance Evaluation. *Journal of Machine Learning Research*, 11:2079–2107, 2010. Article.

[11] National Institute of Standards and Technology. Sign Test. *Dataplot Reference Manual*, updated 2023. Technical reference.

[12] Sture Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979. Article.

Supplemental Analyses

All analyses use the same sixty completed searches. Core reproduction code is available in the public GitHub repository. Prompts, policy sources, and additional analysis inputs accompany this manuscript locally; the reproduction guide identifies additions outside the pinned public release.

S1 Candidate populations during search

S1.1 Population and winner endpoints

I first compared candidate and selected-winner scores in the first and final generations to distinguish progress in newly generated programs from retention of strong incumbents. For each task, I pooled evaluated candidates across ten searches and calculated their median search score. Separately, I calculated the median score of the ten selected generation winners. Scores average returns on ten fixed search episodes; discarded candidates without scores are excluded.

	Candidates		Winners	
Environment	Gen. 1	Gen. 10	Gen. 1	Gen. 10
`CartPole-v1`	21.00	500.00	500.00	500.00
`MountainCar-v0`	-200.00	-120.25	-120.30	-114.90
`Acrobot-v1`	-500.00	-82.70	-82.00	-74.55
`LunarLander-v3`	-423.92	125.80	-32.79	258.04
`BipedalWalker-v3`	-106.43	-30.77	-39.60	-7.58
`CarRacing-v3`	-60.03	555.07	226.05	775.30

Table S1: Candidate and selected-winner medians at generations one and ten.

New-candidate medians increased in all six environments. Selected-winner medians increased in five, as `CartPole-v1` winners were already at the ceiling. These results show progress in the programs being generated beyond retention of a strong early winner.

Generation one contains only fresh proposals, whereas later generations mix fresh proposals, edits, and remixes. The population changes include this composition change. This analysis does not isolate an operator effect or establish held-out improvement of the full candidate population.

S1.2 `CarRacing-v3` distribution across generations

I also examined the full `CarRacing-v3` candidate-score distribution to determine whether rising medians reflected a broad shift or the disappearance of weak candidates. For each generation, I pooled evaluated scores across ten searches and estimated a Gaussian kernel density with a common bandwidth of 25 return units. Multiplying each density by its evaluated-candidate count gives a band whose area equals that count.

The ten generations are stacked from bottom to top. A colored band's *thickness* gives that generation's count density; its cumulative upper boundary is not the density. All 4,992 evaluated scores are included, with 498–500 candidates per generation. Eight discarded slots have no score. Scores average ten selection episodes, and smoothing extends beyond the observed score range.

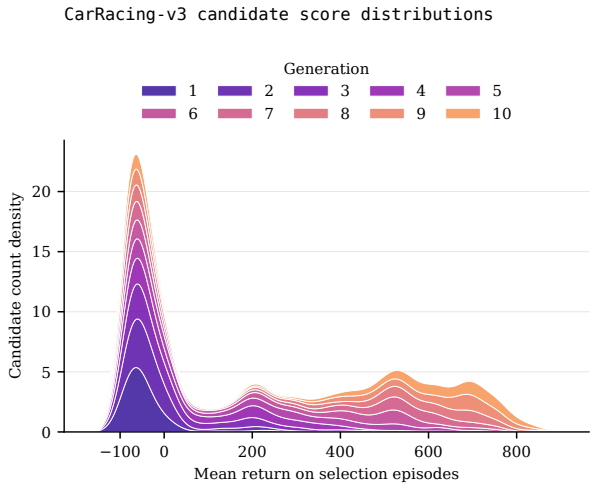

Figure S1: `CarRacing-v3` candidate distributions over ten generations.

Higher-return concentrations emerge and move toward stronger scores as generations advance. Weak candidates persist, but the distribution shifts toward higher returns. These are in-sample changes under a changing operator mixture, not evidence of population-wide held-out gains.

S2 Held-out outcomes and sensitivity

S2.1 Paired search endpoints

I evaluated each search's first and final selected winner on the same 100 held-out episodes. Winners were selected exclusively by search scores; held-out returns did not affect selection. Pairing within each search makes improvement and regression visible without pooling reward scales across tasks.

Hollow circles mark the first generation and filled points mark the final generation. Rightward movement indicates improvement; dashed lines mark reporting targets. Panel headings give the numbers of improved searches and final target-reaching winners. Each task retains its own reward scale.

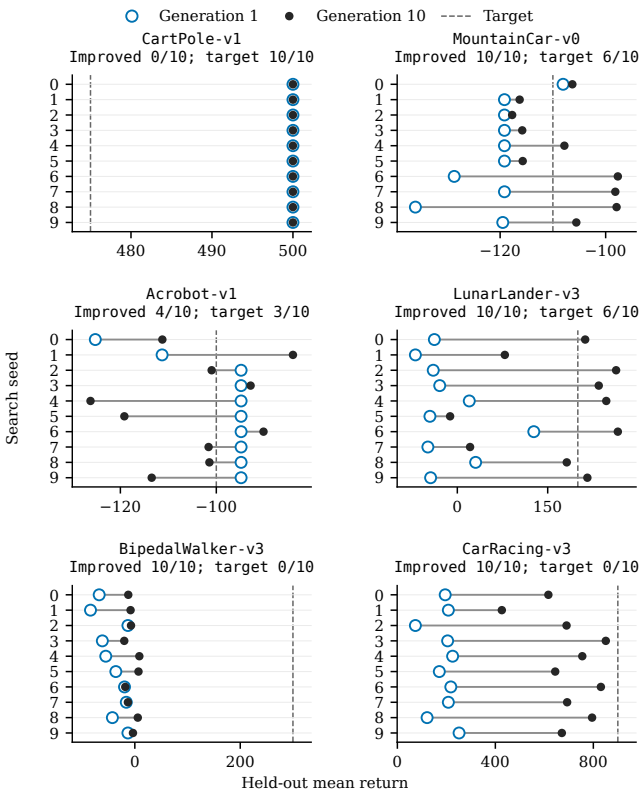

Figure S2: First and final held-out performance for all sixty searches.

All ten endpoint pairs improved in `MountainCar-v0`, `LunarLander-v3`, `BipedalWalker-v3`, and `CarRacing-v3`. `CartPole-v1` remained at the ceiling, while six of ten `Acrobot-v1` searches declined. `BipedalWalker-v3` and `CarRacing-v3` improved without a final winner reaching the reporting target.

All sixty endpoint pairs are available. The missing `MountainCar-v0` search-seed-8 panel at generation six affects only the intermediate trajectory.

S2.2 Sensitivity to individual searches

For each task, I computed the ten paired held-out changes from generation one to generation ten. I omitted one search at a time and recomputed the median of the remaining nine changes. The minimum and maximum of these ten medians measure sensitivity to an individual search; they are not confidence intervals.

CP, MC and AC denote `CartPole-v1`, `MountainCar-v0` and `Acrobot-v1`; LL, BW and CR denote `LunarLander-v3`, `BipedalWalker-v3` and `CarRacing-v3`. The final rows give the full-sample median and leave-one-search-out (LOO) range.

(a) Classic-control tasks

Search seed	CP	MC	AC
0	0.00	1.76	13.98
1	0.00	2.88	27.28
2	0.00	1.46	-6.14
3	0.00	3.37	1.98
4	0.00	11.36	-31.40
5	0.00	3.46	-24.34
6	0.00	31.01	4.67
7	0.00	21.01	-6.78
8	0.00	38.13	-6.61
9	0.00	13.94	-18.64
All ten: median	0.00	7.41	-6.38
LOO min.	0.00	3.46	-6.61
LOO max.	0.00	11.36	-6.14

(b) Box2D tasks

Search seed	LL	BW	CR
0	250.34	55.18	421.20
1	148.46	76.03	218.29
2	303.52	5.89	617.04
3	263.97	41.56	645.68
4	227.37	63.71	529.35
5	33.48	42.99	473.15
6	139.49	1.45	612.59
7	70.05	3.06	484.87
8	151.24	48.18	673.49
9	260.20	9.58	419.54
All ten: median	189.31	42.27	507.11
LOO min.	151.24	41.56	484.87
LOO max.	227.37	42.99	529.35

Table S2: Paired held-out changes and leave-one-search-out medians.

Omitting any one search leaves the direction of every task's median change unchanged. This check concerns variation among the ten searches under common episode panels. It does not establish robustness to new episode panels, prompts, or models.

Endpoint gains also need not be monotonic. At least one adjacent-generation decline occurred in one `CartPole-v1`, five `MountainCar-v0`, ten `Acrobot-v1`, ten `LunarLander-v3`, seven `BipedalWalker-v3` and nine `CarRacing-v3` searches. `LunarLander-v3`'s median fell from 179.38 at generation six to 168.29 at generation seven before reaching 214.04. `CarRacing-v3` fell from 683.81 at generation seven to 658.00 at generation eight before reaching 691.81.

S3 Resource accounting

I aggregated model-call logs, candidate repair counters, discarded slots, and recorded attempt durations across ten searches per task. All sixty searches began at generation zero and finished in one recorded attempt; none resumed a previous search.

For reward-scale context, I also evaluated a random-action policy on the 100-episode held-out panel and report the median of its run-level mean returns. This is not a budget-matched program-search baseline. Calls, repairs, discarded slots (Disc.), and hours are summed across searches. Hours include reporting as well as search.

Task	Random	Calls	Repairs	Disc.	Hours
`CartPole-v1`	21.87	7134	2134	17	4.56
`MountainCar-v0`	-200.00	6157	1157	0	6.05
`Acrobot-v1`	-500.00	6546	1546	9	4.60
`LunarLander-v3`	-200.81	6269	1269	3	4.05
`BipedalWalker-v3`	-101.02	6064	1064	4	7.87
`CarRacing-v3`	-32.68	6119	1117	8	29.76
Total	—	38289	8287	41	56.88

Table S3: Random-action returns and recorded resources by task.

The logs contain 38,289 model calls, including 8,287 repair calls. Repairs account for approximately 21.6% of logged calls, so candidate-slot counts understate model usage. Of 30,000 slots, 29,959 (99.9%) reached evaluated status and 41 were discarded.

Recorded attempt durations total 56.88 hours, including 29.76 hours for `CarRacing-v3`. These are neither CPU-hours nor elapsed wall time for the whole study. Two logged calls lack usage records, so complete token and monetary totals remain unknown.

S4 Operator distributions and elite admission

S4.1 `CarRacing-v3` candidate distributions

I grouped `CarRacing-v3` candidates by generation operator to examine how fresh proposals, edits, and remixes contributed to the score mixture. Generations two, six, and ten represent early, middle, and final post-initialization search, pooling ten searches at each generation.

For each operator and generation, I estimated a Gaussian kernel density with bandwidth 25 return units and multiplied it by the evaluated-candidate count. Band thickness gives count density; band area gives that operator's candidate count. Shared axes and bandwidth support comparison. Counts appear above each panel; the discarded slot at generation six is excluded. Smoothing extends beyond observed returns.

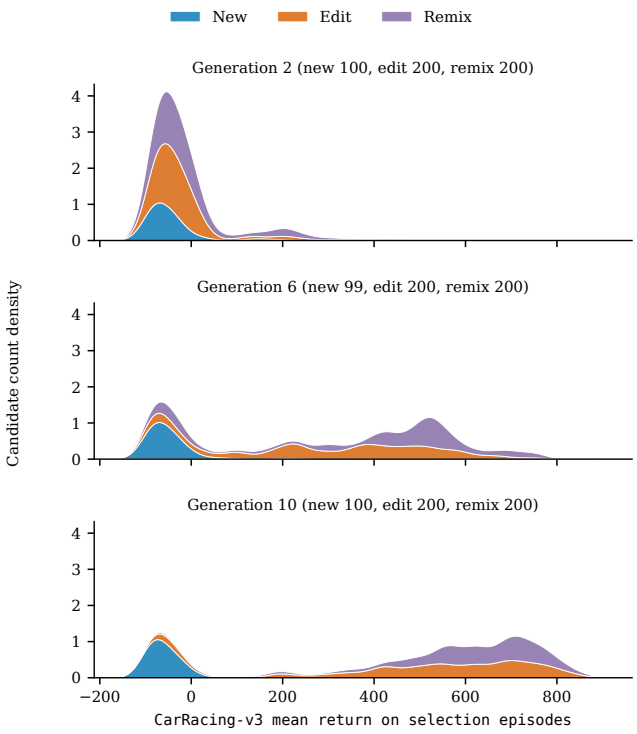

Figure S3: `CarRacing-v3` score distributions by generation operator.

All three operators contribute to the early low-score concentration. By generation ten, fresh proposals remain concentrated at low scores while edits and remixes supply most of the higher-score population. Unequal allocation, parent selection, and inherited information prevent attributing these differences solely to operator choice.

S4.2 Admission to the elite table

I extended the analysis by measuring how often an evaluated candidate entered the elite table in its birth generation, pooling generations 2–10 across ten searches per task. I grouped admissions by the operator used to generate the candidate. Dividing admissions by each operator's evaluated-candidate count accounts for unequal allocations. This measures entry into the table, not the duration of later retention.

Points give admission percentages, with admitted and evaluated counts noted alongside. Discarded slots are excluded from the rate denominators.

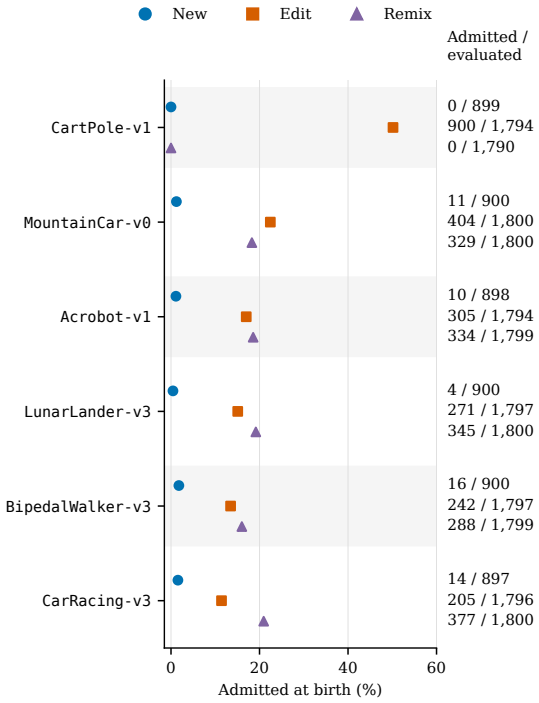

Figure S4: Birth-generation elite-admission rates by task and operator.

Only 55 of 5,394 evaluated fresh proposals entered the elite table, approximately 1%. Fresh proposals had lower observed admission rates under the tested configuration. They receive elite summaries but no parent source, so they are not a history-independent sampling baseline.

Incumbent competition and deterministic tie-breaking also affect admission. `CartPole-v1`'s equal-score priority for edits is especially visible. The rates do not identify causal operator effects, and candidates are dependent observations within searches.

S4.3 Complete operator counts and returns

I also summarized candidates by operator within each task over generations 2–10. The table records slots, evaluated candidates (Eval.), discarded slots (Disc.), median search returns, and birth-generation elite admissions (Adm.). Each search return averages ten selection episodes. Score medians exclude discarded slots; slot totals include them.

`CartPole-v1`					
Operator	Slots	Eval.	Disc.	Median	Adm.
fresh	900	899	1	33.70	0
edit	1800	1794	6	500.00	900
remix	1800	1790	10	500.00	0
`MountainCar-v0`					
Operator	Slots	Eval.	Disc.	Median	Adm.
fresh	900	900	0	-200.00	11
edit	1800	1800	0	-121.80	404
remix	1800	1800	0	-120.30	329
`Acrobot-v1`					
Operator	Slots	Eval.	Disc.	Median	Adm.
fresh	900	898	2	-500.00	10
edit	1800	1794	6	-84.70	305
remix	1800	1799	1	-81.50	334
`LunarLander-v3`					
Operator	Slots	Eval.	Disc.	Median	Adm.
fresh	900	900	0	-361.13	4
edit	1800	1797	3	68.25	271
remix	1800	1800	0	80.70	345
`BipedalWalker-v3`					
Operator	Slots	Eval.	Disc.	Median	Adm.
fresh	900	900	0	-109.16	16
edit	1800	1797	3	-77.02	242
remix	1800	1799	1	-64.61	288
`CarRacing-v3`					
Operator	Slots	Eval.	Disc.	Median	Adm.
fresh	900	897	3	-66.11	14
edit	1800	1796	4	270.51	205
remix	1800	1800	0	391.17	377

Table S4: Operator counts, median search returns and elite admissions.

Fresh proposals had lower median returns than both edits and remixes in every environment. Admissions count elite membership at the candidate's birth generation, not later retention. These are descriptive associations: incumbent competition, parent selection, and tie-breaking prevent attributing the differences solely to operator choice.