

Article

EvoSort: An Audit Protocol for LLM-Driven Program Search—Correctness, Ablation, and the Cost of Too Few Seeds

Karrar Maher Khudhair^{1,2,*}  and Bareq Maher Khudhair² ¹ School of Telecommunication Engineering, University of Málaga, 29071 Málaga, Spain² Department of Computer Techniques Engineering, Imam Al-Kadhim University College (IKC), Baghdad 14522, Iraq; bareq.maher@iku.edu.iq

* Correspondence: karrarmaher@uma.es

Abstract

EvoSort evolves compiled C++ sorting routines. A language model proposes mutations of a baseline sorter, a per-context upper confidence bound (UCB1) bandit picks which operator to try, and a four-gate harness decides whether a candidate may be timed at all. This paper reports what happened when that system was audited at ten seeds per condition. Two conclusions drawn by an earlier version of this work, one from three seeds and one from a single seed, failed in opposite directions. Conditioning operator selection on the input distribution reaches $2.43\times$ over `std::sort` (standard deviation, SD 1.18) against $2.98\times$ (SD 1.00) for one global UCB1 table shared by every island, losing seven of ten paired seeds, so the conditioning buys nothing we can measure. Generalisation, previously reported as a collapse on the evidence of one champion, holds up across ten: they average $3.51\times$ out of sample, spanning $1.50\times$ to $8.39\times$, none is slower than `std::sort` at ten million keys, and six of the ten beat a plain least-significant-digit (LSD) radix sort. Whether radix beats a champion on one of thirty cells or on twenty-four is settled by the seed alone. Correctness held throughout, with zero failures in 420 measured cells covering eleven evolved programs at up to 100 times the training size. Hand-written baselines lose ground at much the same rate across a tenfold increase in input size, so that decline is not evidence of overfitting.

Keywords: program synthesis; large language models; sorting algorithms; multi-armed bandits; evolutionary computation; automated algorithm design; correctness verification; experimental methodology; reproducibility



Academic Editor: Thomas Hanne

Received: 12 August 2026

Revised: 11 September 2026

Accepted: 16 September 2026

Published: 20 September 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

Sorting is among the most studied problems in computing, and its library implementations are the product of decades of hand tuning [1–4]. Two recent lines of work suggest that machines can still find improvements. AlphaDev discovered shorter fixed-length sorting networks with deep reinforcement learning and shipped them into the LLVM standard library [5]. FunSearch and Evolution of Heuristics paired a language model with an evaluator to evolve combinatorial heuristics that beat human baselines [6,7]. A parallel thread—LearnedSort and the learned-index programme—replaced parts of a sort with a model of the data distribution [8–10].

These results share a shape: a generator proposes, an evaluator scores, and something in the loop learns. They differ in what the generator emits (assembly, Python heuristics, a cumulative distribution function (CDF) model) and in whether the loop adapts its own

search. What none of them does is evolve *compiled, general-purpose sorting code* while a learned policy decides, per data distribution, which kind of edit to try next—and then report honestly whether the result survives contact with data it was not trained on.

EvoSort is our attempt at building such a system. It couples a language model proposer with a per-context upper confidence bound (UCB1) bandit over mutation operators and a four-layer correctness harness. Three of its design decisions carry most of the weight, and each is the sort of thing a reader would otherwise have to reconstruct from the experiments, so they are set out before anything else.

- **The learning component is small, and smaller than we first described it.** What adapts during a run is UCB1 [11], applied to a fixed set of mutation operators, with one independent table per island. No feature vector is involved and nothing is shared between islands, so the design is tabular and per-context rather than contextual in the sense of LinUCB and its relatives [12]. An earlier draft of this work called it a contextual bandit. That was the wrong word, and Section 5.1 reports what happened when we finally measured whether the conditioning was worth anything. None of the machinery of AlphaZero [13] is claimed or used.
- **Every timing is earned.** No candidate reaches the benchmark until it has passed a static screen, been compiled in a sandbox, been run in isolation, and had its output checked against the input as an exact permutation. Sorting routines have been attacked in this way for a long time [14–16], and we borrowed most of our adversarial shapes from that literature.
- **The search and the evaluation share no data.** Optimisation runs on three synthetic distributions at 10^5 keys. The winner is then benchmarked on five held-out distributions and ten real datasets at up to 10^7 keys: the Searching on Sorted Data (SOSD) collection [17,18] and columns from the New York City (NYC) taxi records. The distance between those two settings is the most important quantity in this paper. It is large.

What we contribute is not a component. The proposer, the islands, elitism and crossover are adapted from FunSearch-style search and classical genetic programming, and bandit-driven operator selection has a long history in evolutionary computation that recent work has carried into large language model (LLM)-driven heuristic design [19,20]. What we contribute is an *audit protocol* for this kind of system, and an account of what it did to our own claims.

Concretely, we contribute the following: (1) a protocol that gates every timing behind a four-layer correctness check, evaluates on data the search never saw, isolates the mechanism under test with ablations that hold everything else fixed, and runs enough seeds to separate a result from noise. (2) The application of that protocol to EvoSort, which overturned two of our own conclusions in opposite directions—an apparent benefit from conditioning operator selection on the data distribution, which ten seeds erased, and an apparent failure to generalise, which ten seeds also erased (Sections 5.1 and 5.5). (3) A correctness result that did survive: no failure in 420 measured cells across eleven evolved programs, at up to $100\times$ the training size. (4) A defect in our own verifier, found by the search and closed. (5) Two reproducibility hazards encountered in passing and documented rather than smoothed over: the provider replaced the model behind a stable application programming interface (API) name partway through the study, and repriced it.

2. Related Work

Machine-discovered algorithms. AlphaDev framed the search for a faster sort as a single-player game and trained an AlphaZero-style agent to play it, discovering assembly-level routines that improved on the LLVM sort for small fixed inputs [5]. Its scope is the

strength and the limit: tiny, fixed-size networks, verified by exhaustive or bounded testing, not general-purpose comparison sorts on arbitrary input. EvoSort works one level up, on whole C++ functions that sort inputs of any size, and pays for that generality with a weaker correctness guarantee (testing, not exhaustive proof) and a harder benchmarking problem.

LLMs as program generators. Code-trained language models produce working programs from natural-language prompts [21,22], and the same models make surprisingly good evolutionary operators: they propose edits that resemble the changes a programmer would make [23,24]. In FunSearch and EoH this becomes a search loop over heuristics [6,7], while OPRO goes further and uses the model itself as the optimiser [25]. We keep the FunSearch shape of generate, evaluate and retain the best, but two pieces have been added that neither system carries. One is a compiled-native evaluation path, without which a timing reported against `std::sort` would not mean very much. The other is the bandit that decides where the model is worth spending.

Choosing the operator, and how little of that is ours. Deciding which variation operator to apply next from its measured payoff is adaptive operator selection, a question evolutionary computation has worked on for decades and to which bandit algorithms are a standard answer. Recent work carries it into LLM-driven search directly. LAOS drives operator selection with a language model supplied with optimisation state and credit assignment [19]; TIDE nests an island model inside a tuning loop and schedules its prompt strategies with a UCB-based scheduler [20]. Against these, the mechanism in EvoSort is not new and we no longer present it as such. An earlier version of this paper claimed the per-distribution conditioning as its principal contribution; the ablation in Section 5.1 was built to test that claim and did not support it. What this paper offers instead is the test.

Self-refinement. A model can critique and revise its own output [26,27]. We use a narrow version of it, a “reflect” operator which fires when an island has stalled and which passes the strongest candidates together with their measured timings to a stronger model, asking for a redesign rather than an edit. In our runs it underperformed, for a reason that turned out to be mundane and that we document in Section 5.

Learned sorting and indexing. LearnedSort estimates the empirical CDF of the input and uses it to place keys; its successor handles duplicates [8,9], and a later analysis recasts the whole thing as a learning-augmented sample sort [28]. We took two things from the learned-index programme [10]: a framing, and the SOSD benchmark [17,18] that supplies most of our real data. Where EvoSort departs from that line is in what learns. There, a model sits inside a fixed algorithm. Here, the algorithm itself is what changes.

Foundations. The bandit is UCB1 [11,12], and the reinforcement-learning framing around it follows the standard treatment [29] together with the self-play lineage [13,30,31]. Islands, elitism and crossover are classical genetic programming [32,33]; we changed nothing about them. The proposer is a DeepSeek V4 mixture-of-experts model [34,35], built in turn on the transformer [36] and on instruction tuning [37,38].

3. Method

Figure 1 shows the whole loop. A candidate is one C++ file defining a sort function with the fixed signature

```
template <typename K> void evolved_sort(std::vector<K>& a)
```

Islands specialise to distributions. A bandit picks the mutation operator, the language model writes the edit, and a four-layer harness decides whether the result is allowed anywhere near a stopwatch. The rest of this section walks through those pieces in order.

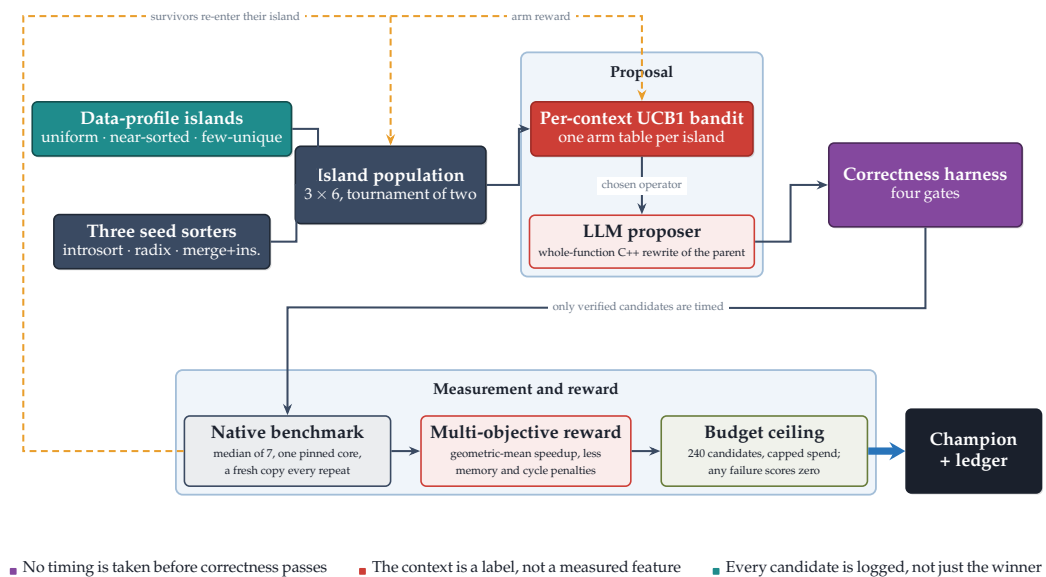


Figure 1. The loop, and the one thing it is not allowed to do. Nothing reaches the benchmark without first clearing the correctness harness, so no timing in this paper was ever taken from an unverified program. Only two paths carry information backwards: surviving candidates return to their island, and a single Bernoulli arm reward returns to the bandit table for the context it came from. Solid arrows show the forward path of a candidate; dashed orange arrows are the two feedback paths. The three notes beneath the diagram are key points rather than a legend, and each marker takes the colour of the stage its note refers to. LLM, large language model; UCB1, upper confidence bound.

3.1. Candidate Representation

A candidate is a complete function. We seed the search with three hand-written sorters (an introsort, a radix-with-insertion hybrid, and a natural merge sort) and let the model rewrite them however it likes. Whole-function generation, in the style of FunSearch [6], gives the search room to restructure control flow, and Section 5 shows that this is where the gains actually come from; a system able only to tune constants would find much less. Against that freedom we set a tight contract: fixed function signature, a whitelist of permitted headers, no I/O, no threads, no global state. Anything outside it is rejected before it costs a compile.

3.2. Islands and Operators

We run one island per training distribution—uniform, near-sorted, and few-unique. Each island’s prompts name its target distribution, so its population specialises, but every candidate is still scored on *all* distributions, so an island cannot win by overfitting to its own focus alone. The best individual migrates to a neighbouring island every 24 candidates, a standard island-model mechanism [32].

The bandit chooses among six operators: `tune_thresholds`, `change_pivot`, `change_partition`, `swap_subsort_dispatch`, `restructure`, and `crossover`. Two further operators are event-triggered rather than chosen: `repair` fires after a candidate fails, feeding the failure text back to the model; `reflect` fires when an island stalls for eight candidates without improvement, handing the two best sorters and their measured timings to a stronger model with a request to redesign rather than tweak.

3.3. Per-Context UCB1

One UCB1 table [11] is instantiated per island, with the six mutation operators as its arms. The island’s declared distribution acts as the context. It is worth being exact about how little that word carries here, because an earlier draft of this work overstated it. The context is a label read from the configuration file rather than a quantity measured from the data. Islands are visited round-robin. No feature vector is ever computed, and the

tables share no parameters, so nothing learned on one island can transfer to another. A feature-based contextual bandit such as LinUCB would do all of those things. This does none of them, and is better described as three independent tabular bandits keyed by a label.

Selection follows the usual rule. An arm that has never been pulled is played first; otherwise the arm maximising

$$\text{UCB}(a) = \hat{\mu}_a + \sqrt{\frac{2 \ln t}{n_a}}, \quad \hat{\mu}_a = \frac{1}{n_a} \sum_{i=1}^{n_a} r_{a,i}, \quad (1)$$

is chosen, where n_a counts the pulls of arm a within that island's table and t the pulls across the table. Both counters are local, so even the exploration term is computed per island.

The reward is Bernoulli. Writing t_{child} and t_{parent} for the median times of a child and its parent *on that island's focus distribution*,

$$r = \mathbf{1}[\text{candidate passes every gate} \wedge t_{\text{child}} < t_{\text{parent}}]. \quad (2)$$

There is no clipping and no ratio; a candidate that fails any gate scores zero, as does one that merely ties. Keeping the reward binary makes UCB1 scale-free and keeps the mechanism short enough to describe in a paragraph, which matters for a component that exists in order to be inspected.

Whether any of this conditioning is worth its complexity is an empirical question, and Section 5.1 answers it by holding everything else fixed and replacing the per-island tables with a single shared one. Figure 2 shows one island's table and the single update each generation makes to it.

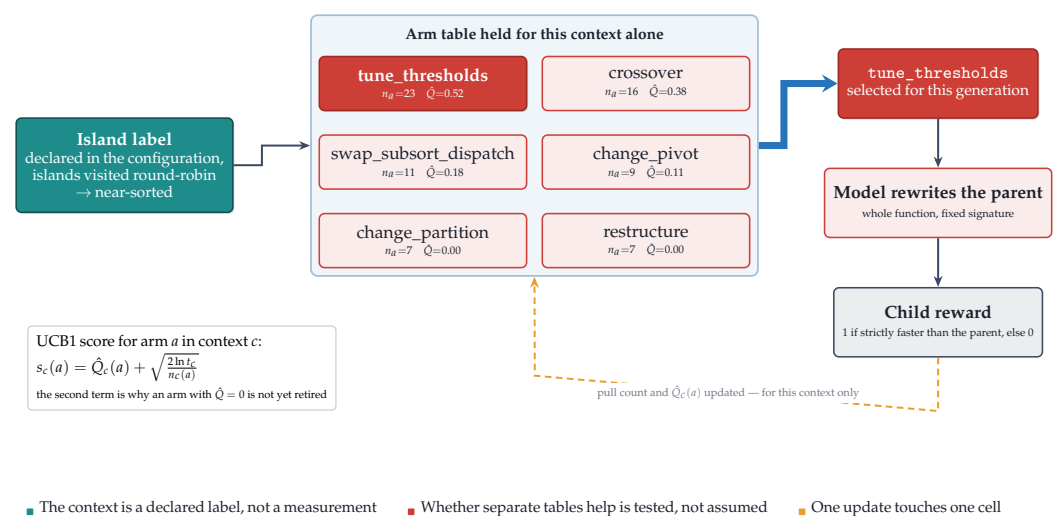


Figure 2. How operator selection works, and what the context label does and does not do. The context is read from the configuration, never measured from the data, and the three tables share nothing, so by the end of a run they disagree about which operator is best. Section 5.1 tests whether that disagreement is worth anything by replacing the three tables with one. Solid arrows follow one generation from the island label, through the operator chosen and the rewrite, to the child's reward; the dashed amber arrow is the single update written back to that island's table. In the table, the operator UCB1 selects is drawn in solid red and the other five are tinted. The notes beneath the diagram are key points, not a legend, and each marker takes the colour of the element its note describes.

3.4. The Correctness Harness

Code produced by a language model is often wrong, sometimes fails to compile, and occasionally corrupts memory, and it does so often enough that the harness was designed

around the expectation rather than around the exception. Four layers stand between a proposal and a measurement anyone should trust, and a candidate that fails at any of them is scored zero.

1. **Static screen.** A textual filter rejects banned constructs before compiling: I/O, threads, inline assembly, non-whitelisted headers, mutable global state. It costs almost nothing, and it hands the repair operator a precise reason for the rejection.
2. **Sandboxed compile.** `g++ -O2 -std=c++20 -static`, with a 60-s timeout. Syntax errors die here, and so do the occasional template expansions that never terminate.
3. **Isolated run.** Each candidate runs in its own subprocess under a wall-clock timeout, so a crash or a hang is recorded as a structured result carrying reward zero. The search survives it.
4. **Adversarial battery.** Before any timing is trusted, the candidate must sort eleven adversarial input shapes—empty, single element, descending pair, all-equal, already sorted, reversed, sawtooth, organ-pipe, two-value, sentinel-laden, and a 200,000-key random array—and must pass, on each, a sortedness check together with an *exact permutation* check against a reference sort. Each shape is instantiated at both 32-bit and 64-bit key widths, so a candidate faces 22 checked executions and must pass all of them. The shapes were drawn from the quicksort-killer and property-based testing literature [14–16].

The exact permutation check is one place where we were unwilling to economise. A candidate that emits sorted-but-wrong output will satisfy a cheap sum-or-xor invariant quite happily, and nothing short of comparison against a reference multiset will catch it. Section 5 reports a case in which that comparison was the only thing standing between us and a bad result.

These gates do real work. Figure 3 gives the attrition of a single run: of 226 proposals, 24 came back without a parseable function, 24 more were stopped by one of the four gates, and 178, or 79%, earned a timing. Every rejection is logged with its cause. Without that record the repair operator would have nothing to work from.

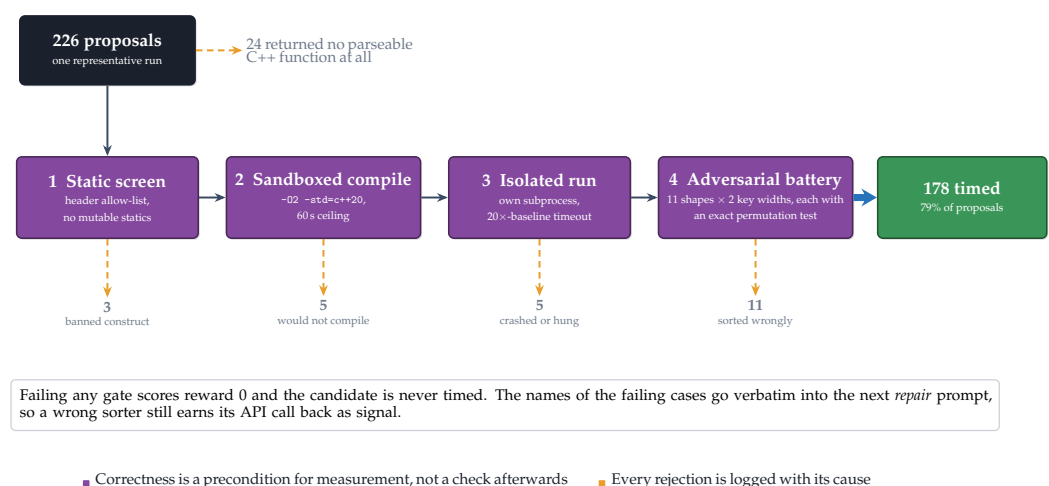


Figure 3. What a model actually gets wrong, counted. The attrition is the seed-1 run’s, read from the **status** field of its ledger. The eleven candidates in the last bucket compiled, ran to completion, and returned output that was not a sorted permutation of the input—which is precisely the class of failure a timing-first pipeline would have measured and rewarded. The dark box holds the run’s proposals, the purple boxes are the four gates in order, and the green box counts the candidates that were timed. Solid arrows carry a candidate forward to the next gate. The dashed amber arrows lead to the candidates each gate rejected, every one of them logged with its cause. The notes beneath the diagram are key points, not a legend, and each marker takes the colour of the element its note describes.

3.5. Reward and Evaluation

A surviving candidate is benchmarked natively: one warm-up plus seven timed repetitions, median reported, on a single pinned core at high priority. We collect wall-clock time, peak resident memory, and thread cycle counts. The last we report as an explicitly-labelled architecture-level energy *proxy*—we do not claim joules, because the hardware Running Average Power Limit (RAPL) counters are not reliably accessible on our platform.

Two different quantities are called a reward in this system, and a reviewer of an earlier draft was right that we never wrote either of them down. Equation (2) gives the first, which the bandit learns from. The second ranks the population and selects the champion:

$$F = w_t S - w_m M - w_c C, \quad (3)$$

where $S = (\prod_{d \in D} t_d^{\text{base}} / t_d^{\text{cand}})^{1/|D|}$ is the geometric-mean speedup over `std::sort` across the training distributions D , and the two penalties are

$$M = \max\left(0, \frac{\text{RSS}_{\text{cand}}}{\text{RSS}_{\text{base}}} - \tau\right), \quad C = \max\left(0, \frac{1}{S_{\text{cyc}}} - 1\right), \quad (4)$$

with S_{cyc} the same geometric mean taken over cycle counts and RSS the peak resident set size. The weights are fixed throughout at $w_t = 1$, $w_m = w_c = 0.25$ and $\tau = 1.5$; they are compiled-in defaults rather than tuned hyper-parameters, and were never varied. Any correctness failure sets F to zero outright.

The two rewards differ in every respect that matters. Equation (2) is binary, compares a child against its *parent*, looks at one distribution, and is the only thing the bandit sees. Equation (3) is continuous, compares a candidate against the *baseline*, aggregates over all three distributions, and never reaches the bandit at all. Because F is built on a geometric mean, a candidate that is excellent on some distributions and worse on others can still score well—a property we return to in Section 6.

3.6. The Proposer and Its Cost

Mutation and crossover are handled by DeepSeek V4-Flash, and the rare reflect calls by DeepSeek V4-Pro [34]. Price decided this for us. A hard per-run spend cap and a duplicate-candidate cache keyed on a hash of the normalised code hold the cost of a 240-candidate run under a dollar.

Quoting a firmer figure than that would be unwise, and the reason is itself worth recording. The flash model was listed at USD 0.14 and 0.28 per million input and output tokens when this work began; on 16 August 2026 the provider replaced those flat rates with peak and off-peak pricing at up to USD 0.44 and 1.32. Our own accounting, built against the older table, understated the eventual invoice by a factor of roughly 2.7. The 33 search runs reported here cost about USD 21 in total. Any cost claim in this area has a date attached to it whether or not its authors say so.

We also ran the loop against a Claude model at a matched 20-candidate budget to check that nothing in the design leans on one provider. Every record in every ledger names the model that produced it.

4. Experimental Setup

Machine and toolchain. All runs and all timings reported here were produced on one machine: an AMD Ryzen 9 3900X (12 cores, 24 threads, 6 MB L2, 64 MB L3), 32 GB RAM, Windows 10 build 19045. The compiler is g++ 16.1.0 (w64devkit, target x86_64-w64-mingw32), C++20, with `-O2` during search and `-O3` for final benchmarks, statically linked, and `-march=native` throughout. That last flag matters for reproduction: every binary is specialised to the host above, so the numbers are not portable to a different

microarchitecture without recompilation. Each run directory records the host specification and compiler version automatically rather than relying on this paragraph.

Timing uses median-of-seven with a warm-up, high process priority, and thread affinity pinned to core 2. Cycle counts come from `QueryThreadCycleTime` and are reported as an architecture-level proxy, not as joules. Full core isolation is not achievable from user space on Windows; we mitigate with the median-of-seven protocol and repeated measurement seeds, and flag the residual noise as a threat in Section 7.

Search configuration. Three islands (uniform, near-sorted, few-unique), population six per island, 240 candidates per run, migration every 24 candidates, reflect after eight-candidate stalls with a cap of six reflects per run. Training inputs are 10^5 unsigned 64-bit keys per distribution.

Held-out and real data. The generalisation benchmark uses five synthetic distributions the search never saw—reverse, sawtooth, organ-pipe, Gaussian, and Zipf—plus ten real datasets: five SOSD collections (books, Facebook, OpenStreetMap (OSM) cell-ids, and Wikipedia timestamps at 200 million keys, and a 32-bit books set) [17] and five columns extracted from the 2024 New York City taxi records (fare, pickup timestamp, pickup location, total amount, trip distance). SOSD files are stored sorted and are deterministically shuffled before timing. We benchmark at 10^6 and 10^7 keys— $10\times$ and $100\times$ the training size—with three measurement seeds.

Conditions. Four search conditions are compared at equal budget, differing only in how the next operator is chosen and identical in every other respect. **Per-context** keeps one UCB1 table per island, which is the system as originally described. **Global** keeps a single table shared by all islands, so adaptive selection is retained and the conditioning is removed; the gap between these two is what the conditioning is worth. **Shuffled** keeps one table per island but permutes which island reads which, holding the number of tables and the data reaching each of them fixed while destroying the meaning of the label; a derangement is forced, so no island can keep its own table. **Uniform** draws operators uniformly at random. Each of the first, second and fourth is run at ten master seeds, the third at three. A fifth arm from earlier work, mutating only by scaling a randomly chosen integer literal and calling no model at all, is quoted once for scale.

Figure 4 lays the split out in full. The separation is the point of the design: three distributions at 10^5 keys shape the objective, and fifteen datasets at 10^6 and 10^7 keys pass judgement on it, with the champion meeting them for the first time after the search has already finished.

Are the timings real? Wall-clock medians taken on a consumer Windows machine deserve some suspicion, so before relying on ours we checked them against an independent counter. Process cycle counts are plotted against median milliseconds for all 420 benchmark cells in Figure 5, and the points fall on a straight line across three orders of magnitude and across all fourteen algorithms. Whatever the scheduler happened to be doing on that machine, it left no visible trace in the medians.

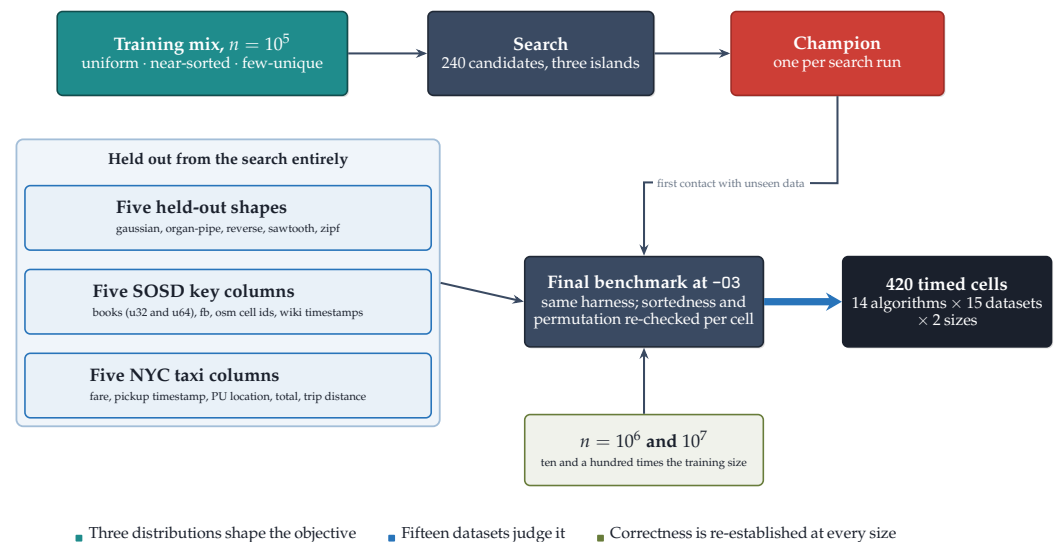


Figure 4. The train/test split, drawn to scale in kind rather than size. Everything the search optimises against sits in the top row. The fifteen evaluation datasets are never sampled, profiled, or tuned against during the run; they meet the champion once, at ten and a hundred times the size it was trained at, through the same harness that verified it. The teal box is the training data, the grey boxes are processing steps (the search and the final benchmark), the red box is the champion a run produces, the blue panel holds the held-out data and the black box is the resulting measurement matrix. Arrows show the direction in which data and programs move; the thick blue arrow marks the step that produces the reported numbers. The notes beneath the diagram are key points, not a legend.

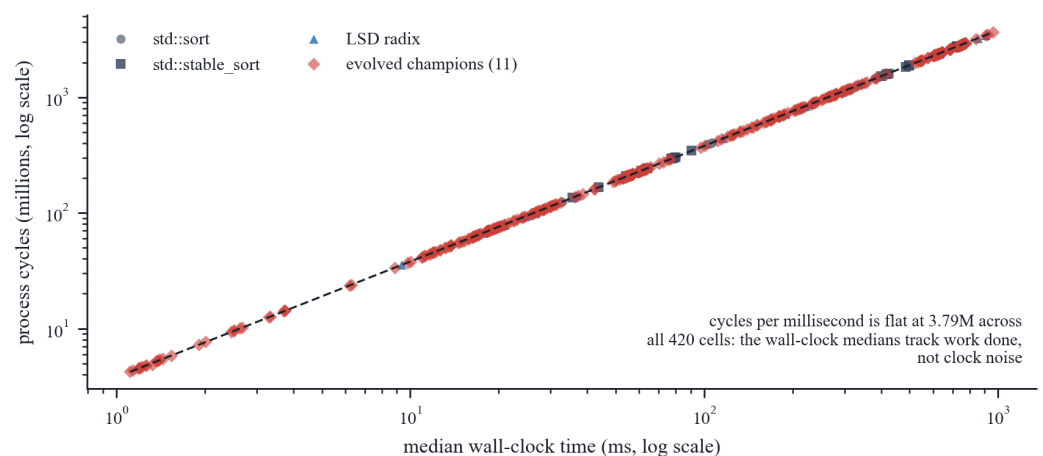


Figure 5. Measurement sanity, all 420 cells. Cycle counts recorded by `QueryProcessCycleTime` are plotted against the median wall-clock time of the same cell, with both axes logarithmic. Any timing distorted by scheduling noise would sit visibly off the line, and none of them does. The dashed line is the pooled ratio of cycles to milliseconds across all cells.

5. Results

5.1. Where the Value Comes from

An earlier version of this work compared exactly two conditions, per-context UCB1 against uniform operator choice, over three seeds. Reviewers of that version made the same objection independently, and it was the right one: such a comparison can establish that UCB1 beats chance, and cannot establish that splitting the table by context contributed anything. The experiment reported here was built to settle that. Four conditions are run, identical in every respect except how the next operator is chosen, at ten master seeds each and three for the label-shuffling control. Table 1 gives the result.

Table 1. Champion reward (geometric-mean speedup over std::sort, three training distributions, in-sample), as mean $\pm$ standard deviation (SD) over master seeds. The three UCB1 variants differ only in how many tables are kept and which one each island reads. Bold marks the highest mean among the conditions run at ten seeds.

Condition	Operator Policy	Seeds	Champion Reward
per-context	one UCB1 table per island	10	2.43 $\pm$ 1.18
global	one shared UCB1 table	10	2.98 $\pm$ 1.00
shuffled	tables permuted across islands	3	3.13 $\pm$ 1.29
uniform	uniform-random choice	10	1.96 $\pm$ 0.61

The conditioning is worth nothing that we can measure. Per-context UCB1 finishes at 2.43 against 2.98 for a single shared table, loses seven of the ten paired seeds, and the difference does not clear a significance test in either direction (Wilcoxon signed-rank, $p = 0.23$). Shuffling the labels so that each island reads a table belonging to a different island costs nothing either; that condition finishes highest of the four, though with three seeds and an SD of 1.29 we read it as noise rather than as a result. What does survive is cruder and older: all three UCB1 variants finish above uniform random choice, by $1.24\times$ for the per-context arrangement and $1.53\times$ for the global one. The gain comes from keeping any statistics on operators at all. It does not come from keeping them separately.

Figure 6 shows why ten seeds were needed to see this. The per-seed points overlap heavily between conditions, and the spread inside a single condition is wider than the gap between conditions. Three draws from these distributions can order them almost any way.

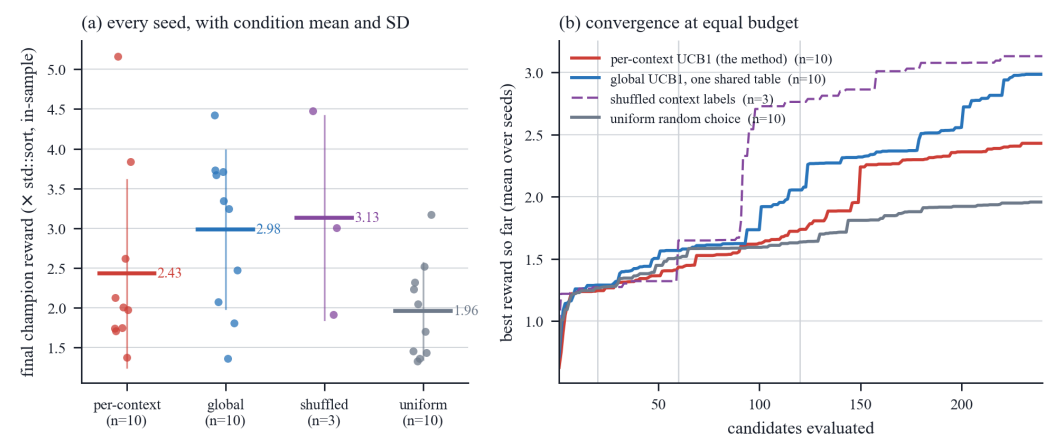


Figure 6. The ablation the first version of this paper was missing. Panel (a) plots every seed, with the condition mean and SD; the shuffled-label control carries three seeds rather than ten and is labelled accordingly. Panel (b) follows the mean best-so-far reward against budget. Per-context UCB1, drawn warm, is the lowest of the three UCB1 variants for most of the run and clears uniform selection only in its final third. In (a), each dot is one seed's champion, the horizontal bar is the condition mean and the vertical line spans one standard deviation either side of it. In (b), each line is the mean over seeds for one condition; the shuffled control is dashed because it rests on three seeds.

The convergence claim from the earlier version does not survive either. That version reported the bandit ahead of uniform selection at half the budget in all three of its seeds, $3.29\times$ against $2.01\times$, and treated arrival time as the effect that lasted. At ten seeds the same measurement gives $1.74\times$ against $1.64\times$ at candidate 120, and at candidates 20 and 60 the per-context arrangement sits fractionally *below* uniform selection, 1.24 against 1.27 and 1.43 against 1.51 . A three-seed study of a process this noisy was reporting its own sampling error.

Cost is not where the constraint lies. A full run finishes for well under a dollar, and the no-LLM arm, which spends nothing and does nothing but jitter integer constants, still

reaches $1.60\times$ (Figure 7). Roughly half of what the model buys is available from blind perturbation. That arm was run once, on the earlier model generation and on different hardware, so it is reported for scale and nothing more; the weakness of this baseline is taken up in Section 7.

Does the choice of model matter? Only a partial answer can be given, and the limits are worth stating before the numbers. Three matched 20-candidate runs exist, one against Anthropic’s Haiku and two against DeepSeek V4-Flash, from the earliest phase of this work (Table 2). At that budget the two providers reach comparable champion rewards while differing by an order of magnitude in price. One of the DeepSeek runs is contaminated: an output-token cap that was too low truncated nine of its twenty generations before any code block closed, which is what its low pass rate records. No full-budget run against Anthropic exists, and none with the bandit enabled, so this table bounds a cost comparison and settles nothing about quality.

Table 2. Matched 20-candidate runs, early in the project. Pass rate counts model-proposed candidates reaching ok. [†] This run was truncated by a token cap since corrected, and is shown rather than dropped because removing an inconvenient run silently is exactly the practice this paper argues against.

Model	Cost	Cost/Cand.	Pass	Champion
claude-haiku-4-5	\$0.203	\$0.0102	16/20	1.25
deepseek-v4-flash [†]	\$0.021	\$0.0011	9/20	1.20
deepseek-v4-flash	\$0.026	\$0.0013	18/20	1.46

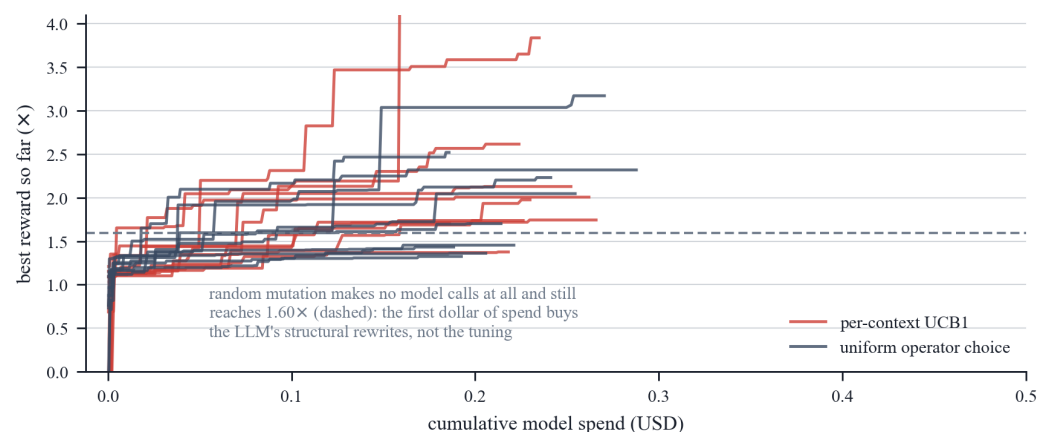


Figure 7. What the search costs. Best reward so far against cumulative model spend, for all six LLM runs; the dashed line is what random constant-jitter achieves for nothing. A full 240-candidate run finishes between USD 0.32 and USD 0.45.

5.2. What the Bandit Learned

Figure 8 shows one run’s finished arm tables (seed 1, the run whose champion we characterise below). They show the bandit doing something sensible. On the near-sorted island, `tune_thresholds` earned a mean arm reward of 0.52 while `restructure` earned 0.00—in this run the bandit learned that tearing up an already-nearly-sorted routine is a waste, and that adjusting its cutoffs is not. On the few-unique island it favoured `crossover` and `swap_subsort_dispatch`; on the uniform island, `restructure` and `tune_thresholds`. Across all operators here, `crossover` was the most reliable—28 of 29 attempts compiled and ran, and 27 beat `std::sort`.

Ten seeds sharpen the caution rather than removing it. Which operator ends up with the highest mean in a given context is not stable across runs, and the estimates behind those means rest on ten to twenty pulls apiece, which is mostly noise. What is stable is

the behaviour one level up: in every seed the tables concentrate their budget on a few operators and push clearly unhelpful ones toward zero instead of spreading pulls evenly. That differentiation is real and uniform selection cannot produce it. Whether differentiating *per island* is better than differentiating once, globally, is the question Section 5.1 answers, and the answer is no.

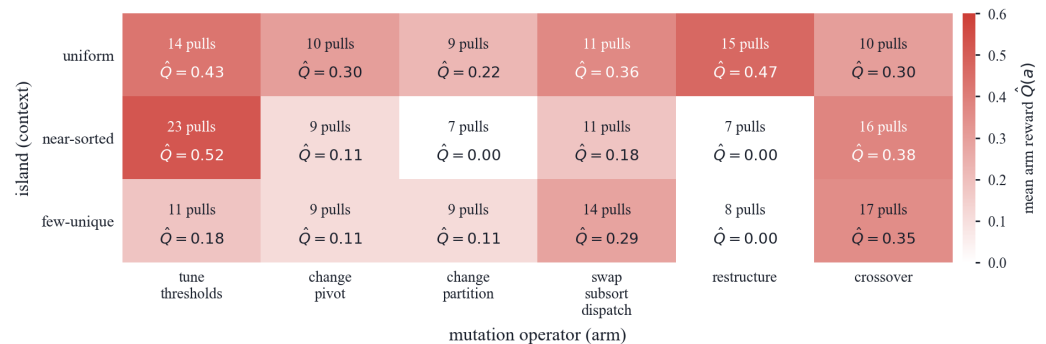


Figure 8. The three islands end the run disagreeing. Pulls and empirical mean arm reward are shown per (context, operator) cell for seed 1. The rows repay being read against one another, since *restructure* is the best arm on uniform input and the worst on near-sorted input, and no single global ranking could hold both of those facts at the same time.

Figure 9 shows how those preferences form as each arm is pulled. Each line is an operator's empirical mean arm reward, $\hat{Q}(a) = \text{successes/pulls}$ —the exact exploitation term UCB1 keeps—as a running average against the number of times that operator was chosen. A first pull lands at 0 or 1 because that is what a Bernoulli mean of one sample is; UCB1's exploration bonus, $\sqrt{2 \ln t / n_a}$, is what stops the bandit from trusting a single lucky or unlucky draw. With nine to twenty-three pulls per arm these curves are directional, not precise.

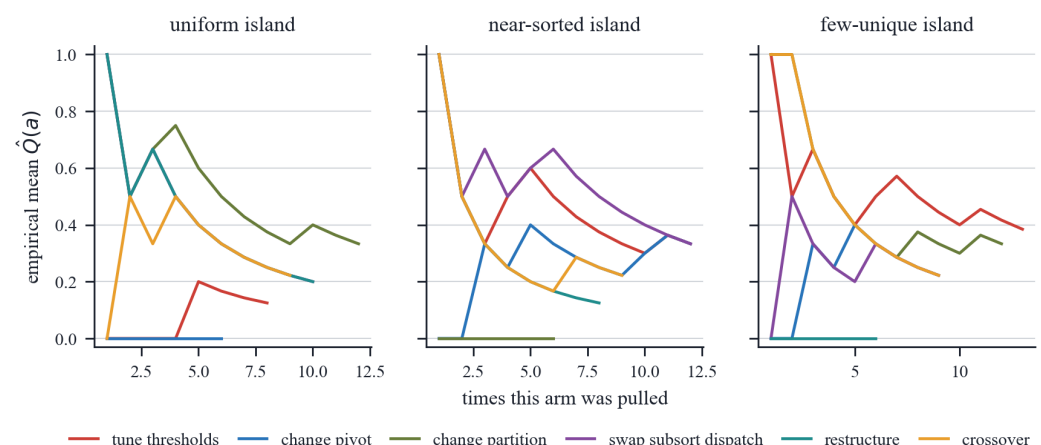


Figure 9. Per-context UCB1 learning curves for seed 2, taken from the recorded per-pull stream with neither reconstruction nor smoothing applied. There is one panel per island, and each line traces one operator's running $\hat{Q}(a)$ against its own pull count, which means the x-axes do not correspond between lines. Arms the bandit has stopped funding appear as flat zeros.

The curves are drawn from seed 2 rather than seed 1, and deliberately so: the per-pull stream that records every bandit update—including the pulls charged as failures when a proposal duplicated an existing candidate—was only added to the logger after the first run, so seed 1 would have to be reconstructed while seeds 2 and 3 can be read directly. We prefer the measured stream to the reconstruction. The cost of that choice is visible if the figure is compared with the heat-map beside it, and it is worth stating rather than hiding: in seed 1

the near-sorted island settled on `tune_thresholds` at 0.52, in seed 2 on `change_pivot` at 0.53, in seed 3 on `crossover` at 0.40. Which arm wins a context is not reproducible at this sample size. What reproduces in all three runs is the shape of every panel: a small number of arms separate upward, at least one is driven to a flat zero and stays there, and the pull budget follows. It is the separation that constitutes the mechanism, while the identity of the winner is noise, and we report both so that a reader can see which is which.

A steadier picture of the operators themselves emerges once all six runs are pooled. Figure 10 counts every proposal each operator produced across the three bandit and three uniform seeds, 1173 of them in total, and records both how often each survived the harness and how good its verified children turned out to be. Mean reward is led by `crossover` and `change_pivot`. The operator worth pausing over is `restructure`, which has the worst survival rate of the six and whose children beat `std::sort` only two-thirds of the time, yet which is responsible for the structural rewrites the champion is built from. An operator that is simultaneously the least reliable and the most valuable is exactly the case a bandit exists to handle.

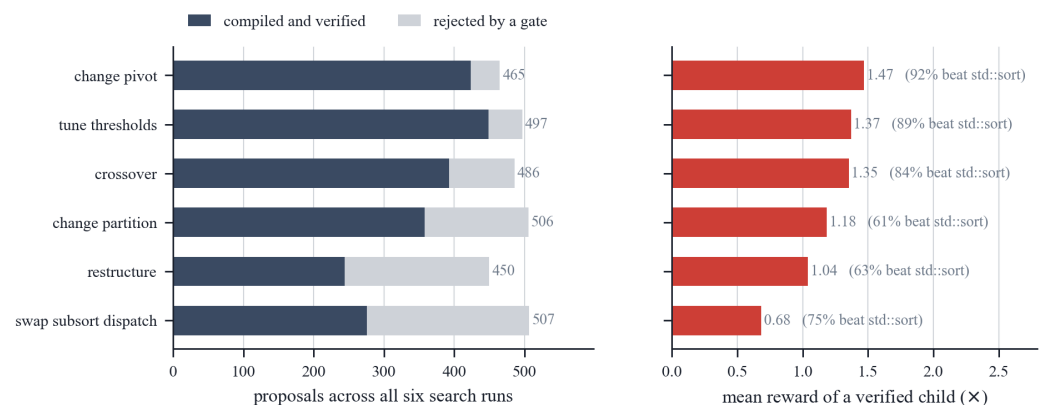


Figure 10. Every operator, pooled over all six search runs. On the left, proposals produced, split by whether the candidate cleared the harness. On the right, mean reward of the verified children, annotated with the share of them that beat `std::sort`. Rows are ordered by mean reward.

The reflect operator failed in our first run for a boring reason. All six reflect calls in seed 1 hit the output-token limit: DeepSeek V4-Pro spends its budget on chain-of-thought before emitting a closing code block, so every reflection came back unusable. We raised the cap fourfold, since the model needs roughly 15,800 output tokens where 8192 had been allowed. Both replication seeds ran with the higher cap and reflect worked there, five of six calls usable in each and four beating `std::sort`. So the champion characterised below came from mutation, crossover and repair alone. The cause is a misconfiguration in that particular run, and says nothing about the operator itself.

5.3. One Champion, Read Closely

This section examines a single evolved program in detail. It is the champion the earlier version of this paper built its conclusions on, kept here as a case study because what it does wrong is instructive, and because Section 5.5 measures how unrepresentative it turned out to be. It comes from the earlier model generation, so nothing about it should be read as typical of the ten reported later.

The sorter is a three-way hybrid, reached through a fourteen-step lineage of `tune`, `pivot`, `crossover`, `restructure` and `repair` edits. Figure 11 shows its dispatch structure, read off the source. It first tries a counting sort when the input has at most 256 distinct keys, detected with a small open-addressed table that aborts on the 257th. Arrays under 256 elements go to a plain introsort. Everything else runs through a 13-bit most-significant-digit bucket sort

followed by a per-bucket introsort. Every piece here is textbook. The search assembled the *composition*, and the search decided to gate the counting-sort path on cardinality; we wrote neither. In-sample the result is $3.64\times$ faster than `std::sort` on uniform data and $13.4\times$ on few-unique data, where the counting-sort path pays off. On near-sorted data it manages only $0.75\times$, and the geometric mean absorbs that regression without complaint.

Two constants in that structure are worth marking. Both gates carry the literal value 256, and neither was ever measured from the input at run time. They are numbers the search settled on against 10^5 -key training inputs and then froze into the program.

Checking the flowchart against the source, as a reviewer asked us to, turned up something we had drawn but not noticed: **the second gate is unreachable**. An array of at most 256 elements cannot hold more than 256 distinct keys, so the counting path's abort on the 257th distinct key can never fire for such an input, and the counting sort always succeeds and returns before the size test is consulted. The introsort branch beneath it is dead code. The search left a vestigial arm in the program and neither the harness, which checks correctness rather than reachability, nor the authors caught it. It costs nothing at run time and it is worth reporting anyway, because it is a small concrete instance of the larger point: evolved code carries structure that nobody designed and nobody reviewed.

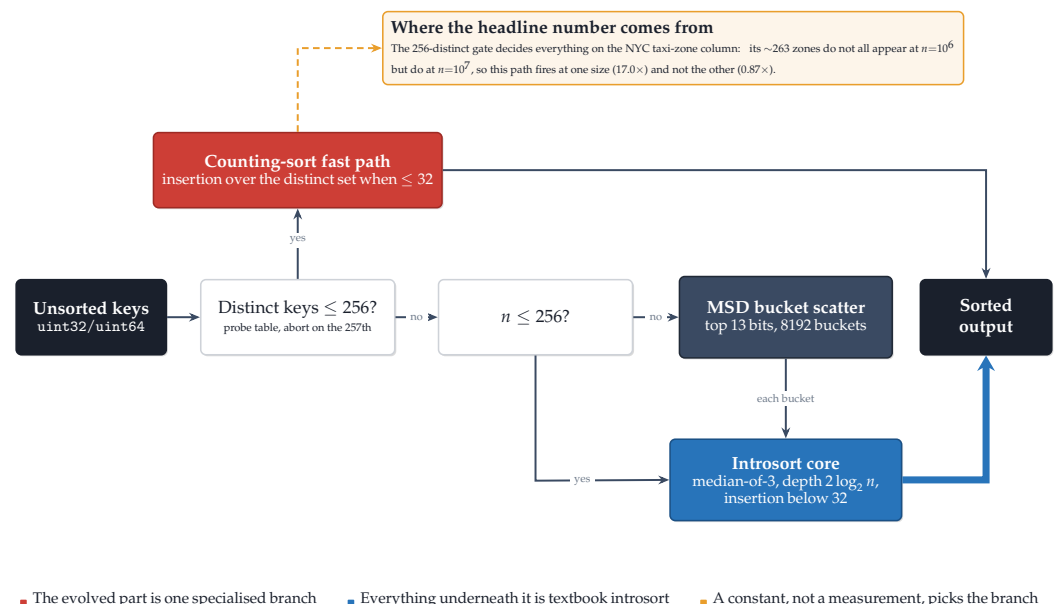


Figure 11. The champion's dispatch tree. What the search contributed is the warm branch together with the constant guarding it, while the other two branches lead to ordinary introsort. Which of the three paths runs is decided by a cardinality test, and that test is answered by a compile-time constant rather than by anything measured from the data at hand. Black boxes are the input and output, white boxes are the two tests, and coloured boxes are sorting stages: red for the evolved counting-sort path, grey for the most-significant-digit (MSD) bucket scatter and blue for the introsort core. Thin arrows are the branches, labelled with the outcome of the test that selects them. The thick arrow marks the path most inputs take. The dashed amber arrow links the counting-sort path to the note above it. The notes beneath the diagram are key points, not a legend.

5.4. The Harness Found Its Own Bug

An earlier champion, from a different run, used a function-local static `std::vector` as a scratch histogram. It passed the adversarial battery—which runs single-threaded and sequentially—but it is not reentrant and not thread-safe: two concurrent sorts would corrupt each other's histogram. Our static screen banned globals and threads but not mutable static locals. The search, in other words, found a correctness hole that our own verifier had missed. We added a rule rejecting mutable static and `thread_local` storage

(while allowing `static const`, and taking care not to trip on `static_cast`), verified that the buggy champion is now rejected at layer one, and confirmed the seed sorters still pass. A search that stress-tests its verifier and exposes a real defect in it is, we think, a result worth reporting on its own.

That episode is only possible because wrong programs are common enough to be a population rather than an anomaly. Figure 12 breaks every proposal in all seven runs down by outcome. Between 74% and 90% of proposals clear the harness in the LLM runs. Among the rest, the category that matters is not the noisiest one—most rejects are simply code that never arrived or never compiled. It is the red band: programs that compiled, ran, terminated, and returned something that was not a sorted permutation of their input. That band is 3% to 6% of every LLM run, and 18% of the no-LLM arm, which jitters constants blindly and so breaks the invariants those constants were holding up. Any pipeline that times first and checks afterwards would have measured every one of them, and would eventually have crowned one.



Figure 12. Every proposal in all seven runs, by outcome. Nine ledger status values are folded into six buckets; the two smallest, a timeout and a provider-side error, are grouped with the failures they are indistinguishable from. The red band is the one that matters: programs that ran to completion and returned the wrong answer.

5.5. Generalisation, and How Much of It the Seed Decides

The earlier version of this paper benchmarked one champion, from one seed, and concluded that evolved sorters do not generalise. Every champion from every seed is benchmarked here instead: eleven evolved programs against `std::sort`, `std::stable_sort` and a tuned single-threaded least-significant-digit (LSD) radix sort, on five held-out distributions and ten real datasets at 10^6 and 10^7 keys. That is 420 measured cells.

Correctness held across all of them. **No failure in 420 cells**, with sortedness and exact permutation verified for every evolved program on every unseen distribution and every real dataset, at up to $100\times$ the size any of them was trained on. Nothing in the harness was relaxed to obtain that. This is the one headline the earlier version reported that ten seeds strengthened rather than removed.

Speed is where the previous conclusion fails. The ten champions of this generation average a geometric-mean speedup of $3.51\times$ out of sample, and the spread around that mean is the finding: from $1.50\times$ to $8.39\times$, SD 2.24. Not one of them is slower than `std::sort` at ten million keys. Six of the ten beat the radix baseline, which itself reaches $2.39\times$. The champion the earlier version reported measures $1.47\times$ here and is the weakest of the eleven. Table 3 breaks these results down by data group.

The same instability reaches the comparison that carried the earlier paper's argument. That version reported radix beating the champion on twenty-seven of thirty cells. Figure 13

counts the same thing for each champion in turn: radix wins anywhere between **one and twenty-four of the thirty cells**, decided by nothing except which seed was drawn, and the earlier champion sits at twenty-five, past the end of the range the ten seeds cover. A conclusion was drawn from a point that turned out to be the extreme of a distribution nobody had sampled.

In-sample reward is a poor guide to any of this. Across the ten champions the rank correlation between training reward and out-of-sample geomean is $+0.48$, and the apparent strength of the Pearson coefficient, $+0.88$, rests on two champions at the top of the range: excluding them, the relationship among the remaining eight reverses to -0.27 while their out-of-sample results still span $1.50\times$ to $3.66\times$. Training performance separates the very best from the rest and says almost nothing within the rest (Figure 14). Figure 15 shows the same spread cell by cell.

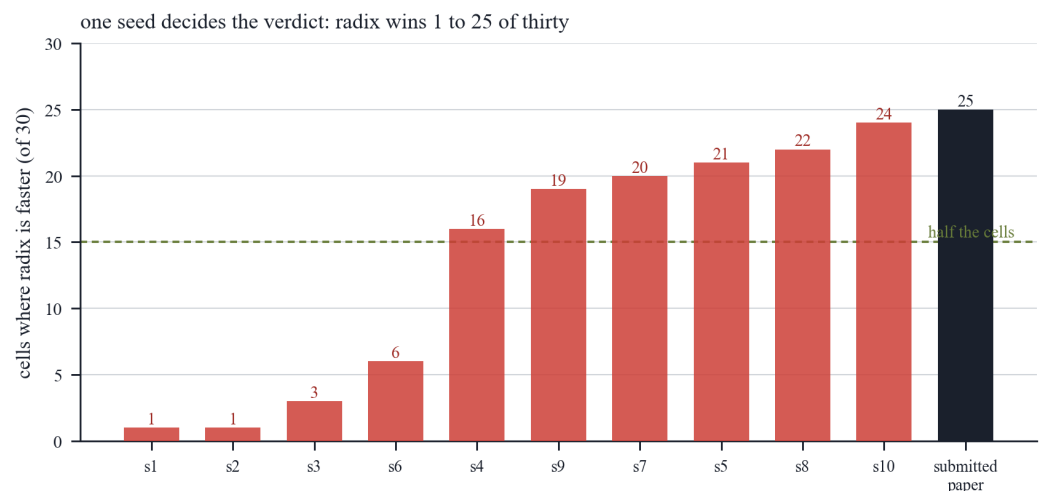


Figure 13. How often a plain radix sort beats the evolved champion, per seed. The earlier version of this paper answered this with a single number, 27 of 30. Across ten seeds the answer runs from 1 to 24, and the champion that version reported, drawn dark, is the least general of the eleven measured.

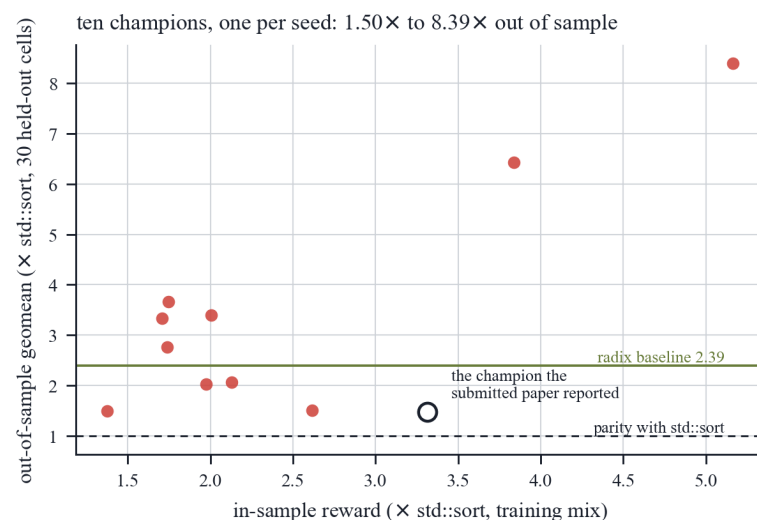


Figure 14. In-sample reward against out-of-sample geometric mean for the ten champions of this generation, with the radix baseline drawn as a reference line and the champion the earlier version reported circled. The vertical spread at any given training reward is what makes a single-champion generalisation study uninformative. Each filled dot is one champion. The solid line marks the radix baseline ($2.39\times$), the dashed line marks parity with `std::sort` ($1\times$), and the open circle is the earlier version's champion.

One further correction belongs here, because the earlier version read a real measurement the wrong way. Every champion does lose ground between 10^6 and 10^7 keys, and that was offered as evidence of overfitting. It is not. Radix keeps 58% of its advantage across the same size step and `std::stable_sort` keeps 73%, against a median 55% for the champions; neither baseline was evolved and neither can have overfitted anything (Figure 16). What the tenfold size step exposes is the memory hierarchy. Any claim that a search has overfitted has to clear that bar first, and ours did not.

What the tenfold size step actually costs is drawn in Figure 16, for every evolved program and both hand-written baselines at once. The lines fall together. Champions keep a median 55% of their advantage, radix keeps 58% and `std::stable_sort` 73%, and since neither baseline was evolved, neither can be losing ground to overfitting. Whatever the earlier version of this paper read into a single champion's decline, the decline itself is a property of the machine rather than of the search.

Table 3. Generalisation on held-out and real data, across all ten champions of this generation. Geomean speedup over `std::sort` per group; the champion column gives the range and median over the ten. Every champion sorts every cell correctly. The last column counts how many of the ten beat the radix baseline on that group. The bold row gives the result over all thirty cells.

Group	Champions Min/Med/Max	Radix	Stable_Sort	Champs >Radix
Held-out synth.	1.18/2.23/7.26	1.54×	0.85×	8/10
SOSD (real)	1.14/2.28/5.17	3.79×	1.03×	4/10
NYC taxi (real)	1.77/3.32/16.52	2.35×	0.96×	8/10
Overall	1.50/3.05/8.39	2.39×	0.95×	6/10

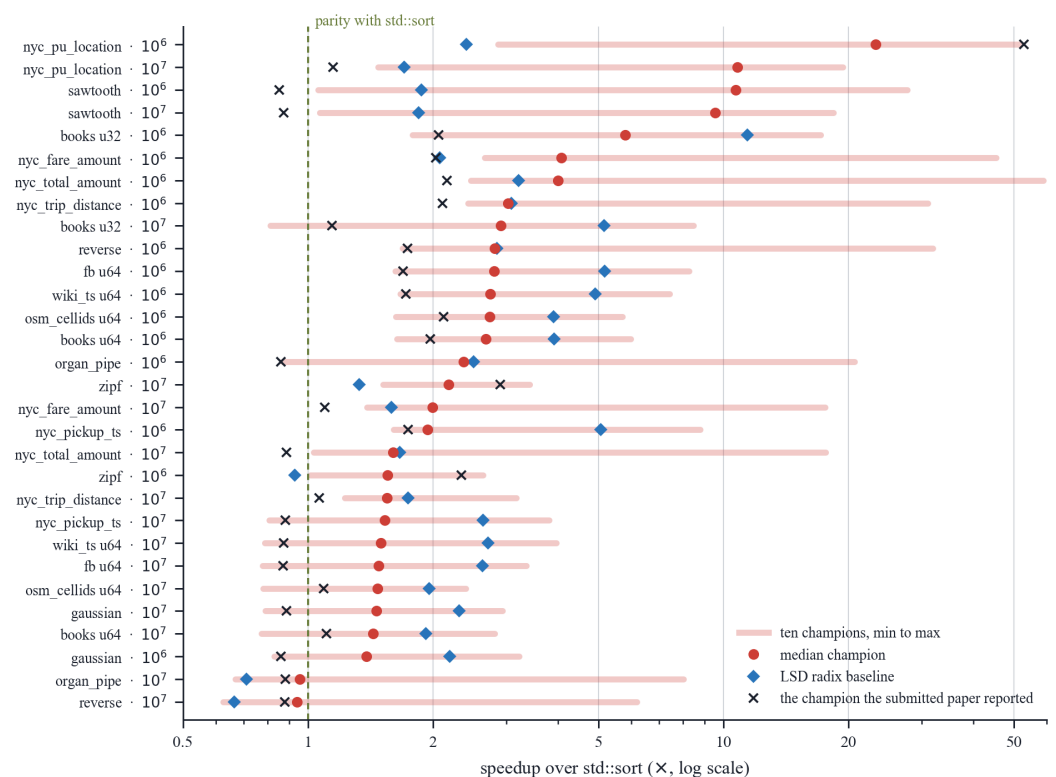


Figure 15. Every unseen cell, ordered by the median champion, on a log scale. The bar spans the ten champions from slowest to fastest, the dot marks their median, the diamond is the radix baseline and the cross is the champion the earlier version of this paper reported. In most rows the bar is wider than the distance between the median and radix, which is the difficulty with drawing any conclusion here from one seed. The dashed vertical line marks parity with `std::sort` ($1\times$).

The earlier version of this paper read that case as textbook overfitting and traced it to a single line of the reward function: a geometric mean dominated by one $13.4\times$ few-unique win, which paid the search to abandon the general-purpose radix sorter already sitting in its own population. Figure 17 sets out that chain, and every number in it is measured.

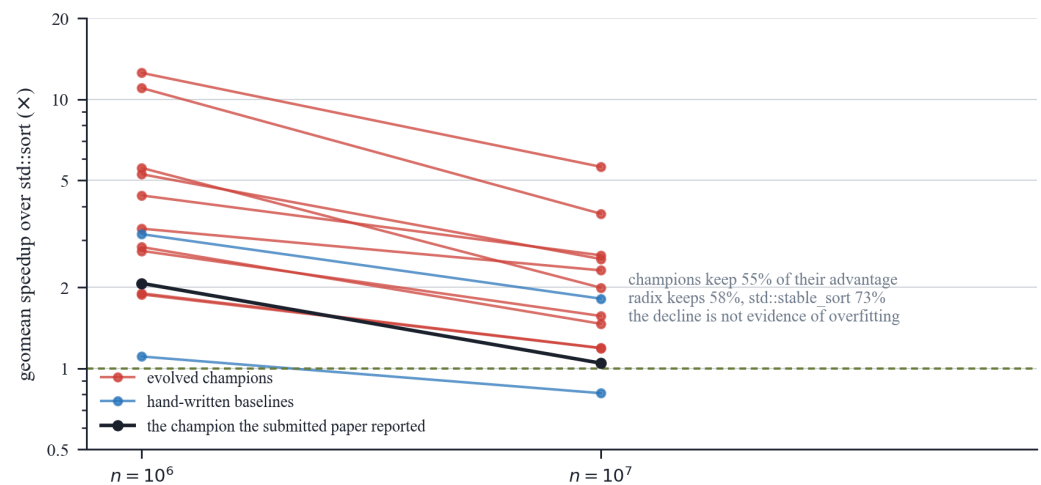


Figure 16. What a tenfold increase in n costs, one line per algorithm, log scale. Every evolved champion is drawn warm, the two hand-written baselines cool, and the champion the earlier version of this paper reported in dark. The baselines were written by people and fitted to nothing, yet they lose ground at much the same rate, which is why this decline cannot be read as evidence of overfitting.

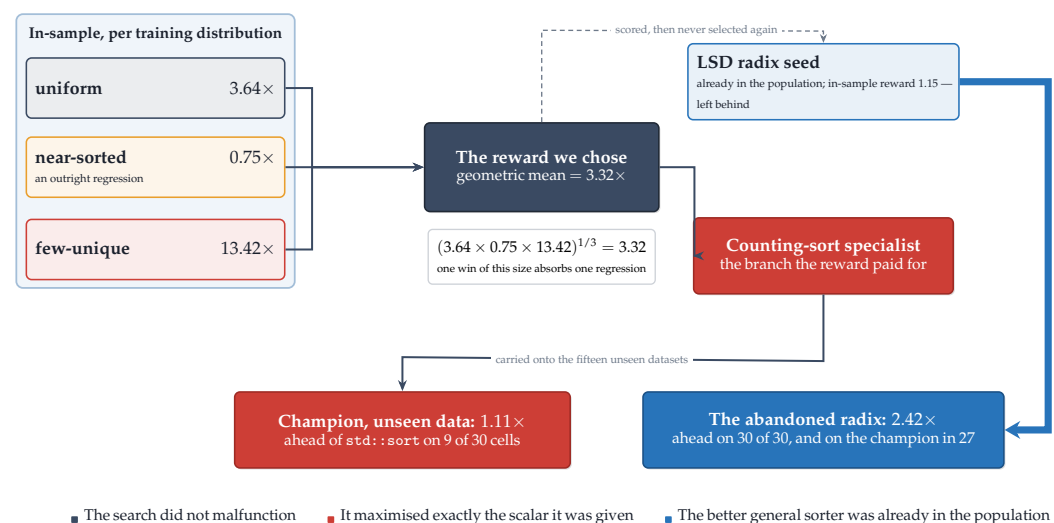


Figure 17. How one run produced a specialist, with every figure measured from that run's own records. This chain holds for the champion the earlier version of this paper reported. It does not generalise: nine further seeds optimising the same objective on the same training mix did not end up here, which is why the objective can be said to permit this outcome but not to cause it. The unseen-data figures in the bottom row come from the earlier version's benchmark. Re-measured in this revision's 420-cell matrix, the same champion reaches $1.47\times$ and radix beats it on 25 of 30 cells (Figure 13). On the left, the three in-sample results sit in tinted boxes (grey for uniform, amber for the near-sorted regression, red for the few-unique win); the grey box is the reward that combines them, and the white box beneath it is the formula. Red boxes follow the specialist the reward selected, and light-blue boxes follow the radix sorter, first as the seed the search left behind and then on unseen data. Solid arrows show how the reward was computed and what it selected; the thin line leads to the radix seed, which was scored and then never selected again. The thick blue arrow follows that seed onto the unseen data. The notes beneath the diagram are key points, not a legend.

We now think the chain is real for this run and much too strong as a general claim, and we withdraw the general version. Nine other seeds, optimising the same objective on the same training mix, produced champions averaging $3.51\times$ out of sample, six of which beat radix. An objective that reliably caused overfitting would not do that. What the objective does is *permit* uneven specialists; whether a given run finds one appears to depend on the draw. The honest reading is that this particular champion overfitted and that we cannot attribute it to the objective alone, since the training mix and the search space are equally available as explanations and we have not separated them.

6. Discussion

What survived. Two claims came through the audit intact. The correctness protocol held: no failure in 420 cells, eleven evolved programs, unseen distributions and real data at up to $100\times$ the training size, and along the way it caught a reentrancy defect in our own verifier that we had not thought to screen for. And keeping any statistics at all on which operator pays off beats choosing blindly, by $1.24\times$ for per-island tables and $1.53\times$ for a single shared one. Cost is not a barrier either: a full run finishes for well under a dollar, which puts this kind of search within reach of anyone.

What did not. Two claims from the earlier version of this work did not survive, and they failed in opposite directions. Conditioning operator selection on the data distribution appeared to help at three seeds and buys nothing measurable at ten. The evolved sorter appeared not to generalise when one champion was measured, and generalises perfectly respectably when ten are. One error flattered us and the other did not, which is the useful part: a systematic bias would show a direction, and this shows only noise.

Why one seed is not a study. The reason both errors were available is visible in Figure 6. Within a single condition the seed-to-seed spread is wider than the distance between conditions, and out of sample the ten champions range from $1.50\times$ to $8.39\times$. Drawing three samples from distributions that overlap this much can order them almost any way, and drawing one says nothing at all. This is not a peculiarity of our system. Program search driven by a language model is stochastic at every level—sampling temperature, which operator fires, which parent is picked—and published work in the area routinely reports one run. Our own two reversals are the argument for not doing that.

What the specialisation tells us. The champions are not worse sorters than `std::sort`; they are *different* ones, and how different depends on the run. On low-cardinality or heavily-skewed data—taxi zone ids, Zipf keys—the better ones are dramatically faster, because the search finds and gates a counting-sort path. Against a plain radix sort the outcome is genuinely mixed: six of ten champions beat it overall, and on the SOSD collection only four do. A practitioner reading this should take the method as producing a specialist whose domain must be checked rather than assumed. Sampling a few thousand keys and counting distinct values costs almost nothing, and where cardinality is small or the data visibly skewed the counting path will pay for itself. Where it is not, a plain radix sort remains a strong and far simpler default.

7. Threats to Validity

Ten seeds, not a hundred. Each main condition was replicated across ten master seeds and the label-shuffling control across three. Ten was enough to retire two conclusions that three and one seed had supported, which is the main lesson we take from it. It is not enough to make small differences significant: the per-context against global comparison gives $p = 0.23$ on a Wilcoxon signed-rank test, so what we report is an absence of measurable benefit rather than a demonstrated disadvantage. The shuffled control at three seeds is

weaker still and we treat its apparent lead as noise. The no-LLM arm was run once, on the earlier model generation and on other hardware, and is quoted only for scale.

The provider changed the model, and its price, mid-study. Partway through this work deepseek-v4-flash began emitting chain-of-thought under an unchanged API name, and the reasoning tokens are billed as output and counted inside the token limit. Runs that had produced a median of 4056 output tokens with 93.5% clean stops began terminating at the cap 100% of the time, producing nothing usable. DeepSeek offers no dated snapshot to pin, so every condition reported here was regenerated under one configuration with thinking disabled, and the earlier generation is retained separately rather than mixed in. The provider also replaced flat pricing with peak and off-peak rates on 16 August 2026. Any cost figure in this area needs a date attached to it, and any comparison spanning a provider’s silent model swap is confounded whether or not its authors noticed.

Comparing across model generations. For that reason the champion the earlier version of this paper reported cannot be compared cleanly against the ten reported here: it came from a model version that no longer exists, so seed and model are confounded in that particular comparison. The claims we defend live inside one generation, where the model, the configuration and the hardware are held fixed and only the seed varies.

The no-LLM baseline is weak. It perturbs integer literals and nothing else, while the model restructures whole functions and performs crossover. The comparison therefore bounds what blind constant-jitter achieves; it does not isolate the value of the language model. A structured non-LLM mutation operator would be the right control and we did not build one.

Measurement noise. Full core isolation cannot be had from user space. Median-of-seven timing on a pinned core reduces scheduling noise without eliminating it, so differences under 5% should be read as ties. Candidates are compiled with `-march=native`, which makes every binary specific to the host; the host specification is recorded automatically in each run directory for that reason.

The objective is still a geometric mean. It rewards uneven wins, and we have not changed it. Our earlier reading, that this objective caused the generalisation failure, does not survive the ten-seed data and has been withdrawn; the more cautious statement is that the objective permits uneven specialists, and that per-distribution reporting is a mitigation rather than a fix.

Baseline scope. The strong non-comparison baseline is a tuned single-threaded LSD radix sort. We do not compare against `ips4o` [39], a state-of-the-art in-place parallel sample sort, because it does not build cleanly under our MinGW toolchain. A parallel baseline would be faster still, so the comparison flatters the champions if anything.

Scale. We benchmarked to 10^7 keys, not the full 200 million in the SOSD files. Cache behaviour beyond 10^8 may differ again.

8. Conclusions

EvoSort couples a language-model proposer, a per-context UCB1 bandit and a strict correctness harness into a loop that evolves compiled C++ sorters for well under a dollar a run. What we can defend after auditing it properly is narrower than what we first believed, and more useful. The harness works: no failure in 420 cells across eleven evolved programs at up to $100\times$ the training size, and it surfaced a reentrancy defect in itself. Keeping statistics on which operator pays off beats choosing blindly. Conditioning those statistics on the input distribution does not, and the evolved sorters generalise considerably better than a single-champion study had suggested.

The lesson is about method rather than about sorting. Two of this project’s conclusions came from too few seeds, and both dissolved at ten—one in the flattering direction and one

in the other. Program search driven by a language model is stochastic in several places at once, its run-to-run spread routinely exceeds the effects being claimed, and a literature that reports one or three runs is reporting its own sampling error a good deal of the time. The remedy is dull and costs only compute: run enough seeds to see the distribution, ablate the mechanism you are claiming credit for against a version that removes only that mechanism, and benchmark on data the search never saw. We would rather be a worked example of that than of the alternative.

What comes next follows from the same place. The training mix should be designed for coverage rather than for a convenient geometric mean; a structured non-LLM mutation baseline would isolate what the model actually contributes; and the reentrancy defect the search surfaced points toward co-evolving the verifier alongside the sorter, so that the thing doing the checking grows harder as the thing being checked grows cleverer.

Author Contributions: Conceptualisation, K.M.K. and B.M.K.; methodology, K.M.K. and B.M.K.; software, B.M.K.; validation, B.M.K.; formal analysis, K.M.K. and B.M.K.; investigation, B.M.K.; data curation, B.M.K.; writing—original draft preparation, K.M.K. and B.M.K.; writing—review and editing, K.M.K.; visualisation, B.M.K.; supervision, K.M.K.; project administration, K.M.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding. The language-model API charges for the thirty-three search runs reported here came to approximately USD 21, paid by the authors.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All thirty-three search runs are retained in full. Each is a directory holding the per-candidate ledger, which records for every model call its identity, token counts, stop reason and running cost, together with the per-pull bandit stream, the resolved configuration, the captured host specification and the champion source. The 420-row benchmark matrix, the evolutionary loop, the correctness harness, the exact prompt templates and every analysis and figure script are retained alongside them under `evosort/`, `scripts/` and `results/`. Every number in this paper traces to a named run directory and to the script that produced it. The runs behind the earlier version of this work are kept unmodified so that the two generations can be compared. These materials are available from the corresponding author on request. The SOSD collection and the 2024 New York City taxi records are third-party datasets available from their original sources.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Hoare, C.A.R. Quicksort. *Comput. J.* **1962**, *5*, 10–16. [\[CrossRef\]](#)
2. Sedgewick, R. Implementing Quicksort programs. *Commun. ACM* **1978**, *21*, 847–857. [\[CrossRef\]](#)
3. Musser, D.R. Introspective Sorting and Selection Algorithms. *Softw. Pract. Exp.* **1997**, *27*, 983–993. [\[CrossRef\]](#)
4. Sanders, P.; Winkel, S. Super Scalar Sample Sort. In *Proceedings of the 12th Annual European Symposium on Algorithms (ESA)*; LNCS; Springer: Berlin/Heidelberg, Germany, 2004; Volume 3221, pp. 784–796. [\[CrossRef\]](#)
5. Mankowitz, D.J.; Michi, A.; Zhernov, A.; Gelmi, M.; Selvi, M.; Paduraru, C.; Leurent, E.; Iqbal, S.; Lespiau, J.B.; Ahern, A.; et al. Faster sorting algorithms discovered using deep reinforcement learning. *Nature* **2023**, *618*, 257–263. [\[CrossRef\]](#) [\[PubMed\]](#)
6. Romera-Paredes, B.; Barekatin, M.; Novikov, A.; Balog, M.; Kumar, M.P.; Dupont, E.; Ruiz, F.J.R.; Ellenberg, J.S.; Wang, P.; Fawzi, O.; et al. Mathematical discoveries from program search with large language models. *Nature* **2024**, *625*, 468–475. [\[CrossRef\]](#) [\[PubMed\]](#)
7. Liu, F.; Tong, X.; Yuan, M.; Lin, X.; Luo, F.; Wang, Z.; Lu, Z.; Zhang, Q. Evolution of Heuristics: Towards Efficient Automatic Algorithm Design Using Large Language Model. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*; PMLR: Cambridge, MA, USA, 2024; Volume 235, pp. 32201–32223.
8. Kristo, A.; Vaidya, K.; Çetintemel, U.; Misra, S.; Kraska, T. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*; ACM: New York, NY, USA, 2020; pp. 1001–1016. [\[CrossRef\]](#)
9. Kristo, A.; Vaidya, K.; Kraska, T. Defeating duplicates: A re-design of the LearnedSort algorithm. *arXiv* **2021**, arXiv:2107.03290.

10. Kraska, T.; Beutel, A.; Chi, E.H.; Dean, J.; Polyzotis, N. The Case for Learned Index Structures. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data*; ACM: New York, NY, USA, 2018; pp. 489–504. [\[CrossRef\]](#)
11. Auer, P.; Cesa-Bianchi, N.; Fischer, P. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.* **2002**, *47*, 235–256. [\[CrossRef\]](#)
12. Lattimore, T.; Szepesvári, C. *Bandit Algorithms*; Cambridge University Press: Cambridge, UK, 2020. [\[CrossRef\]](#)
13. Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* **2018**, *362*, 1140–1144. [\[CrossRef\]](#) [\[PubMed\]](#)
14. Claessen, K.; Hughes, J. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*; ACM: New York, NY, USA, 2000; pp. 268–279. [\[CrossRef\]](#)
15. McIlroy, M.D. A killer adversary for quicksort. *Softw. Pract. Exp.* **1999**, *29*, 341–344. [\[CrossRef\]](#)
16. de Gouw, S.; Rot, J.; de Boer, F.S.; Bubel, R.; Hähnle, R. OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In *Proceedings of the Computer Aided Verification (CAV)*; LNCS; Springer: Cham, Switzerland, 2015; Volume 9206, pp. 273–289. [\[CrossRef\]](#)
17. Kipf, A.; Marcus, R.; van Renen, A.; Stoian, M.; Kemper, A.; Kraska, T.; Neumann, T. SOSD: A Benchmark for Learned Indexes. NeurIPS Workshop on Machine Learning for Systems. *arXiv* **2019**, arXiv:1911.13014.
18. Marcus, R.; Kipf, A.; van Renen, A.; Stoian, M.; Misra, S.; Kemper, A.; Neumann, T.; Kraska, T. Benchmarking Learned Indexes. *Proc. VLDB Endow.* **2020**, *14*, 1–13. [\[CrossRef\]](#)
19. Zhang, Y.; Yi, G. LAOS: Large Language Model-Driven Adaptive Operator Selection for Evolutionary Algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*; ACM: New York, NY, USA, 2025; pp. 517–526. [\[CrossRef\]](#)
20. Chen, C.; Zhong, M.; Fan, Y.; Shi, J.; Sun, J. TIDE: Tuning-Integrated Dynamic Evolution for LLM-Based Automated Heuristic Design. *arXiv* **2026**, arXiv:2601.21239. [\[CrossRef\]](#)
21. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374.
22. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-level code generation with AlphaCode. *Science* **2022**, *378*, 1092–1097. [\[CrossRef\]](#) [\[PubMed\]](#)
23. Lehman, J.; Gordon, J.; Jain, S.; Ndousse, K.; Yeh, C.; Stanley, K.O. Evolution through Large Models. *arXiv* **2022**, arXiv:2206.08896.
24. Meyerson, E.; Nelson, M.J.; Bradley, H.; Gaier, A.; Moradi, A.; Hoover, A.K.; Lehman, J. Language Model Crossover: Variation through Few-Shot Prompting. *ACM Trans. Evol. Learn. Optim.* **2024**, *4*, 1–40. [\[CrossRef\]](#)
25. Yang, C.; Wang, X.; Lu, Y.; Liu, H.; Le, Q.V.; Zhou, D.; Chen, X. Large Language Models as Optimizers. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, Vienna, Austria, 7–11 May 2024.
26. Madaan, A.; Tandon, N.; Gupta, P.; Hallinan, S.; Gao, L.; Wiegrefe, S.; Alon, U.; Dziri, N.; Prabhumoye, S.; Yang, Y.; et al. Self-Refine: Iterative Refinement with Self-Feedback. In *Proceedings of the Advances in Neural Information Processing Systems 36 (NeurIPS)*; NeurIPS: San Diego, CA, USA, 2023.
27. Shinn, N.; Cassano, F.; Berman, E.; Gopinath, A.; Narasimhan, K.; Yao, S. Reflexion: Language Agents with Verbal Reinforcement Learning. In *Proceedings of the Advances in Neural Information Processing Systems 36 (NeurIPS)*; NeurIPS: San Diego, CA, USA, 2023.
28. Carvalho, I.; Lawrence, R. LearnedSort as a learning-augmented SampleSort: Analysis and Parallelization. In *Proceedings of the 35th International Conference on Scientific and Statistical Database Management (SSDBM)*; ACM: New York, NY, USA, 2023. [\[CrossRef\]](#)
29. Sutton, R.S.; Barto, A.G. *Reinforcement Learning: An Introduction*, 2nd ed.; MIT Press: Cambridge, MA, USA, 2018.
30. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533. [\[CrossRef\]](#) [\[PubMed\]](#)
31. Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. Mastering the game of Go without human knowledge. *Nature* **2017**, *550*, 354–359. [\[CrossRef\]](#) [\[PubMed\]](#)
32. Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*; MIT Press: Cambridge, MA, USA, 1992.
33. Real, E.; Aggarwal, A.; Huang, Y.; Le, Q.V. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the AAAI Conference on Artificial Intelligence*; AAAI Press: Palo Alto, CA, USA, 2019; Volume 33, pp. 4780–4789. [\[CrossRef\]](#)
34. DeepSeek-AI. DeepSeek-V4: Towards Highly Efficient Million-Token Context Intelligence. *arXiv* **2026**, arXiv:2606.19348.
35. DeepSeek-AI. DeepSeek-V3 Technical Report. *arXiv* **2024**, arXiv:2412.19437.
36. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention Is All You Need. In *Proceedings of the Advances in Neural Information Processing Systems 30 (NeurIPS)*; NeurIPS: San Diego, CA, USA, 2017; pp. 5998–6008.
37. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. In *Proceedings of the Advances in Neural Information Processing Systems 33 (NeurIPS)*; NeurIPS: San Diego, CA, USA, 2020; pp. 1877–1901.

38. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.L.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. In *Proceedings of the Advances in Neural Information Processing Systems 35 (NeurIPS)*; NeurIPS: San Diego, CA, USA, 2022; pp. 27730–27744.
39. Axtmann, M.; Witt, S.; Ferizovic, D.; Sanders, P. In-Place Parallel Super Scalar Samplesort (IPS4o). In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA)*; LIPIcs; Schloss Dagstuhl—Leibniz-Zentrum für Informatik: Wadern, Germany, 2017; Volume 87, pp. 9:1–9:14. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.