SMALL BUT MIGHTY: QUALITY–DIVERSITY EVOLUTIONARY SEARCH FOR SIZE-OPTIMIZED UNIT TEST GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Reliable coding agents require verification signals that are not only strong but cheap enough to execute repeatedly within agentic loops. Yet existing unit-test generation methods optimize coverage or mutation score while leaving suite size uncontrolled, often producing increasingly redundant tests. We formulate *size-optimized unit test generation* as a quality-diversity problem: maximize behavioral coverage while discovering compact suites along the quality–size frontier. We introduce TESEARCH, an inference-time evolutionary search that partitions test suites into size niches via MAP-Elites and maintains a Pareto frontier over coverage and mutation score within each niche. A *size-annealing UCB* allocates generation budget across niches, prioritizing quality early and progressively favoring smaller suites once quality is secured. Crucially, quantitative control through an LLM is itself stochastic: requesting k tests does not reliably produce k tests. TESEARCH therefore models the LLM as a noisy actuator through an online *landing kernel* and compensates for its size-instruction bias before generation. Across 129 TestGenEval modules and three frontier LLMs, TESEARCH outperforms conventional inference-time scaling and coding agents, achieving higher hypervolume with substantially smaller suites. The optimized suites further improve test-guided cross-version migration and provide fast, effective acceptance checks for self-evolving agents.

1 INTRODUCTION

Coding agents are rapidly becoming capable of long-horizon, repository-level software engineering (Anthropic, 2025; OpenAI, 2025; DeepSeek-AI, 2026). As agents increasingly write, modify, and even evolve software autonomously, a central bottleneck shifts from *generation* to *verification*: how can an agent determine whether a proposed change preserves the intended behavior of a system? In software engineering, the executable answer is the test suite. Here, tests serve as executable *regression specifications* (Yoo & Harman, 2012): they capture the accepted behavior of existing code so that subsequent modification, migration, or modernization can preserve it.

For an agentic loop, however, a useful test suite must be not only *effective*, but also *efficient*. High coverage and mutation score indicate that a suite exercises and distinguishes program behavior, but every additional test increases execution cost, context length, and maintenance burden. This cost compounds when the suite is executed after every agent action, software revision, or self-modification. Existing test-generation methods largely optimize effectiveness while leaving suite size uncontrolled (Fraser & Arcuri, 2011; Lukasczyk & Fraser, 2022). LLM-based methods generate human-readable tests, while inference-time refinement improves their effectiveness through executable feedback but does not optimize suite size, often accumulating tests as optimization proceeds (Chen et al., 2021; Alshahwan et al., 2024; Jain et al., 2025). Conversely, post-hoc reduction can only shrink an existing suite, potentially sacrificing behavioral coverage. What is missing is a method that directly searches for suites that are both *strong and small*.

We formulate this problem as *size-optimized unit test generation*: given executable quality signals, discover test suites that maximize behavioral coverage while minimizing the number of tests required to achieve it. A naive approach would scalarize quality and size into a single objective. This

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

creates an undesirable trade-off: larger suites are systematically favored by coverage-based objectives, while a strong size penalty can compress a suite before sufficient behavior has been captured. Instead, we make a different choice: **size defines where a suite competes, while quality defines whether it wins**. We treat suite size as a behavioral descriptor and cast test generation as a quality–diversity (QD) problem (Fontaine & Nikolaidis, 2021). The search space is partitioned into size niches, and test suites compete only against suites of comparable size. This preserves high-quality candidates across the size spectrum rather than collapsing search toward a single operating point.

Building an effective search around this formulation introduces two challenges. First, LLM calls are expensive, so a finite inference-time budget must be allocated across size niches whose potential value is initially unknown. The search must explore different suite sizes, exploit niches that yield useful tests, and eventually favor smaller suites without sacrificing quality. Second, unlike behavioral descriptors in classical QD, suite size cannot be directly controlled: the algorithm can only request a target size from the LLM. However, LLMs do not reliably satisfy such numerical constraints (Zhou et al., 2023); requesting k tests may yield a suite with a different number of tests. Thus, the niche targeted by a generation may differ from the niche in which the generated suite actually falls. Ignoring this mismatch leads to incorrect budget allocation and credit assignment.

We address these challenges with TESEARCH, a quality–diversity evolutionary search for size-optimized test generation. Following MAP-Elites (Mouret & Clune, 2015), TESEARCH partitions the search space into geometric size niches, each maintaining a Pareto front over coverage and mutation score. A *size-annealing UCB* allocates LLM calls across niches, balancing exploration and exploitation while progressively favoring smaller suites once quality is secured. Because LLMs do not reliably follow size instructions (Zhou et al., 2023), an online *landing kernel* models the discrepancy between requested and realized suite sizes, correcting systematic bias and making allocation aware of where each generation is likely to land. Finally, TESEARCH selects parents by size proximity and crosses them using executable coverage and mutation feedback.

We evaluate TESEARCH on 129 real-world Python modules from TestGenEval (Jain et al., 2025), under a matched inference budget and across three frontier LLM backbones. TESEARCH consistently outperforms existing inference-time scaling methods and coding agents, achieving higher hypervolume with substantially smaller suites. The resulting suites further transfer beyond the optimization objective, improving acceptance oracles for cross-version migration and providing stronger, more efficient verification for self-evolving coding agents. To summarize, our contributions are as follows:

- We formulate *size-optimized unit test generation* as a quality–diversity problem, treating suite size as a behavioral descriptor rather than scalarizing it with test quality.
- We introduce **TESEARCH**, which combines a size-niched multi-objective archive with a *size-annealing UCB* that allocates inference-time compute across the quality–size search space and shifts toward compact suites only after quality is secured.
- We model quantitative LLM generation as *noisy actuation* and introduce an online *landing kernel* that compensates for size-instruction bias and incorporates the resulting uncertainty into search.
- Across 129 TestGenEval modules and three frontier LLMs, TESEARCH produces stronger and smaller suites than existing inference-time scaling methods and coding agents, with downstream gains in cross-version migration and verification for self-evolving agents.

2 Related Work

Test generation. Traditional search-based software testing generates tests by evolutionary search over program inputs (Fraser & Arcuri, 2011; Lukasczyk & Fraser, 2022); LLMs replaced the generator with a model that writes human-idiomatic tests directly (Chen et al., 2021), and industrial deployments report measurable coverage gains at scale (Alshahwan et al., 2024). TestGenEval (Jain et al., 2025) established a real-world benchmark measuring generated suites by coverage and mutation score (Jia & Harman, 2011). This line optimizes quality at whatever size emerges. The complementary line of *test-suite reduction* shrinks an existing suite post hoc under a coverage-preservation constraint (Harrold et al., 1993; Yoo & Harman, 2012), but subtractive minimization demonstrably sacrifices fault-detection power (Rothermel et al., 2002), and it can only choose among tests that were already written. Controlling size *during* generation remains unaddressed by either line.

Test-time search, evolutionary search and quality–diversity optimization. *Test-time search* iteratively optimizes textual context against downstream feedback (Yang et al., 2024a; Feng et al., 2025), with methods such as MIPROv2 (Opsahl-Ong et al., 2024), GEPA (Agrawal et al., 2026),

and ACE (Zhang et al., 2026) demonstrating systematic improvement through reflection and feedback accumulation. Test generation naturally fits this paradigm because its feedback is *executable*: generate a suite, measure coverage and mutation score, and refine. However, existing methods optimize a fundamentally different object. They optimize a single textual artifact, whereas a test suite is a variable-size collection whose quality is defined collectively and whose cardinality is itself an optimization target. Evolutionary search with LLMs as variation operators and executable feedback as fitness has emerged as a powerful inference-time optimization paradigm (Romera-Paredes et al., 2024; Lange et al., 2026; Novikov et al., 2025). However, optimizing a global objective tends to concentrate the population around a single mode (Goldberg & Richardson, 1987; Deb et al., 2002). Quality–diversity (QD) optimization instead partitions the search space by *behavior descriptors*, allowing quality to be optimized independently across diverse regions (Lehman & Stanley, 2011a;b; Cully & Demiris, 2018). MAP-Elites maintains an elite per region (Mouret & Clune, 2015), while multi-objective extensions such as MOME maintain Pareto fronts and aggregate their hypervolumes (Pierrot et al., 2022). We cast test generation as QD search with suite size as the descriptor. Unlike classical QD, however, size must be *requested* from an LLM before it can be observed, and the requested and realized sizes need not match. TESEARCH explicitly models this actuation error with an online landing kernel, enabling search over a descriptor that is only imperfectly controllable.

3 Problem Formulation

We study *size-optimized unit test generation*: given a source module, generate a test suite that maximizes the collective quality of the suite while minimizing its cardinality. Given a Python module M, produce test suites for M. A *suite* T is a finite set of test functions for M, with size $s(T)$. Two objectives are measured on T: *coverage* $c(T) \in [0, 1]$, the fraction of executable lines of M executed, and *mutation score* $m(T) \in [0, 1]$, the fraction of a fixed sample of mutants of M (Jia & Harman, 2011) killed. Each suite is thus a point $(c(T), m(T))$ in quality space. With the origin as reference, a suite has hypervolume $\mathrm{hv}(T) = c(T) \cdot m(T)$ (Zitzler & Thiele, 1999), which is high only when both objectives are high.

Definition 1 (Size-optimized unit test generation)**.** *Given a module M and executable quality metrics (c, m), produce a collection of test suites $\mathcal{T}$ and return the smallest suite among those achieving maximal hypervolume:*

$$\mathcal{T}_{\max} = \arg\max_{T \in \mathcal{T}} \mathrm{hv}(T), \qquad T^\star \in \arg\min_{T \in \mathcal{T}_{\max}} s(T). \tag{1}$$

Quality–diversity formulation. Equation 1 defines the *terminal objective*: maximize quality first, then minimize size among equally qualified suites. Rather than scalarizing quality and size, we use quality–diversity as a *search surrogate*: size defines *where* a suite competes, while coverage and mutation score define *how well* it performs. Because larger suites are naturally advantaged in global quality competition (Namin & Andrews, 2009; Inozemtseva & Holmes, 2014), we treat size as a behavioral descriptor and optimize quality independently within size niches. This yields a MAP-Elites formulation (Mouret & Clune, 2015), with each niche maintaining a Pareto front over (c, m) rather than a single elite (Pierrot et al., 2022).

The size-niched bins. Concretely, the archive keeps a Pareto front per niche,

$$\mathcal{A}_j \;=\; \underset{(c,m)}{\mathrm{nd}} \big\{\, T \in \mathcal{T} \;:\; b_{j-1} \le s(T) < b_j \big\}, \qquad j = 1, \ldots, K, \tag{2}$$

where nd takes the non-dominated subset on (c, m) and $b_0 < \cdots < b_K$ are the niche boundaries, niche j holding the sizes in $[b_{j-1}, b_j)$ and the outermost two left open ($b_0 = 0$, $b_K = \infty$) so that every suite falls in exactly one niche. Size determines only *which* niche a suite competes in, not its quality. We use geometric niches because the variation in generated suite size grows with the requested size, so niches are spaced by equal ratios rather than equal numbers of tests (§A).

Search loop and challenges. The search iterates over the size-niched archive (Figure 1): select, generate, and file back. A target niche is selected, and one parent is taken from each side of its target size, the Pareto-front suite nearest in log size across the entire archive (refer to §D for details). Conditioned on these parents, the LLM generates a suite toward the target size; the suite is executed, scored on (c, m), and filed into its realized niche if non-dominated. This loop raises two challenges. First, *allocation*: under a finite LLM-call budget, the search must decide which size niches to explore when their potential gains are unknown. Second, *actuation*: the LLM only approximately follows requested suite sizes, so targeted and realized niches may differ, confounding allocation and credit assignment. §4 addresses both.

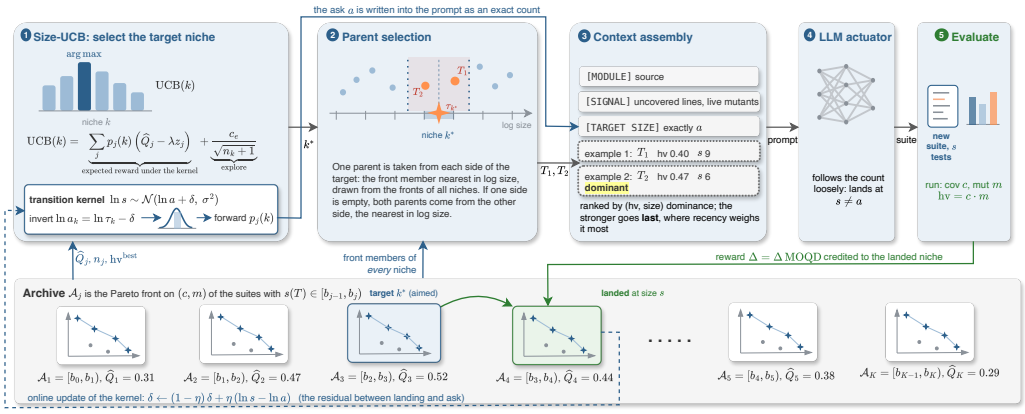


Figure 1: **Overview of TESEARCH.** The archive (bottom) partitions suites by size, with each niche storing a Pareto front of multiple non-dominated suites under coverage and mutation score. Each iteration: **(1)** Size-UCB selects a target niche using a transition kernel that accounts for imperfect LLM size control; **(2)** the Pareto-front suite nearest the target size on each side is selected as a parent; **(3)** the parents, module, execution feedback, and target size form the generation context; **(4)** the LLM generates a suite, possibly at a different size than requested; and **(5)** the suite is evaluated and filed into its realized niche, and the size error updates the transition kernel.

4 TESEARCH

TESEARCH addresses the two challenges with two components. First, a *transition kernel* (§4.1) models how the realized suite size deviates from the requested size, allowing TESEARCH to correct this bias and predict where each generation will land. Second, a *size-annealing UCB* (§4.2) allocates the generation budget across size niches using these predictions, while progressively favoring smaller suites once sufficient quality is achieved. Given the selected niche, §E describes how parents and execution feedback are assembled into the generation context.

4.1 THE TRANSITION KERNEL ON THE ACTION SPACE

LLMs follow size instructions only approximately: requesting a suite of size a may produce one of size $s \neq a$. We therefore distinguish the *requested size* a from the *realized size* s, which determines where the suite lands in the archive. This mismatch complicates allocation: a call targeting one niche may land in another, making target-based UCB credit assignment inaccurate.

We model the mismatch between requested and realized suite sizes in log space, since generation variability scales with the requested size (§3). We divide the niches geometrically (Masson et al., 2019; van Erven & Koolen, 2016), following the intuition that LLMs follow count instructions with multiplicative noise (Zhang et al., 2025; Li et al., 2024; Xiao et al., 2026). We model landing in log space with the Gaussian approximation

$$\ln s \sim \mathcal{N}\big(\ln a + \delta, \sigma^2\big), \tag{3}$$

where δ captures systematic size bias and σ captures residual variation. This provides a tractable operational model of multiplicative size noise for predicting landing probabilities, rather than an exact distributional assumption. §5.3 evaluates its empirical fit.

The two parameters play different roles. The bias δ varies across modules and directly determines the size requested from the LLM, so we estimate it online and compensate for it before generation. In contrast, σ controls only the uncertainty over which niche a generation may reach; it is stable across modules and is therefore fixed to the median measured on the holdout set (§G.3.1).

Online aiming and update. Given a target niche k with center $\tau_k = \sqrt{b_{k-1} b_k}$, we first choose the request a_k so that its expected landing is centered at τ_k. Inverting (3) gives

$$\ln a_k = \ln \tau_k - \delta, \tag{4}$$

where a_k is rounded to an integer before prompting the LLM. Thus, the current bias estimate δ is compensated for *before* generation.

Because the realized size remains stochastic, a request aimed at niche k may land in another niche. For each niche j with boundaries $[b_{j-1}, b_j)$, its landing probability is

$$p_j(k) = \Phi\left(\frac{\ln b_j - \mu_k}{\sigma}\right) - \Phi\left(\frac{\ln b_{j-1} - \mu_k}{\sigma}\right), \qquad \mu_k = \ln a_k + \delta. \tag{5}$$

Here, $p_j(k)$ is the probability of landing in niche j when targeting niche k. The resulting distribution $\{p_j(k)\}_{j=1}^{K}$ is used by the UCB (§4.2) to evaluate each target niche according to where its generation is likely to land.

After observing the realized size s, we update the bias estimate using the LMS delta rule (Widrow & Hoff, 1988):

$$\delta \leftarrow (1-\eta)\delta + \eta\big(\ln s - \ln a\big), \tag{6}$$

where $\eta \in (0, 1]$ controls the bias adaptation rate. This online update learns the module-specific size bias, allowing subsequent requests to better target the desired suite size.

4.2 SIZE-UCB: WHICH NICHE RECEIVES THE NEXT CALL

We cast niche selection as a multi-armed bandit: each arm k is one niche of the archive, pulling arm k spends one LLM call on a generation aimed at that niche, and the reward is what that call adds to the collection. What it adds to is the multi-objective quality-diversity score (Pierrot et al., 2022), the total area the archive's fronts dominate. Let $\mathcal{A} = \{\mathcal{A}_1, \ldots, \mathcal{A}_K\}$ denote the size-niched archive. Its multi-objective quality–diversity (MOQD) score is

$$\mathrm{MOQD}(\mathcal{A}) = \sum_{j=1}^{K} \mathrm{HV}(\mathcal{A}_j), \qquad \mathrm{HV}(\mathcal{A}_j) = \mathrm{Area}\left(\bigcup_{T \in \mathcal{A}_j} [0, c(T)] \times [0, m(T)]\right), \tag{7}$$

which reduces to the single-suite hv of §3 when a front holds one member. Thus, MOQD measures the total dominated objective-space area across all size niches. A generated suite improves the archive if it expands the Pareto front of its realized niche, including when it populates an empty niche. We define the reward of a call as the increase in MOQD caused by inserting its generated suite: $\Delta = \mathrm{MOQD}(\mathcal{A}^{+}) - \mathrm{MOQD}(\mathcal{A})$. The reward captures both improvement of an existing niche and population of an empty niche, and is credited to the niche where the suite actually lands. Because the archive retains only non-dominated suites, $\Delta \geq 0$ and is bounded.

The acquisition balances exploitation, exploration, and suite size. For a target niche k, we average niche values over its landing distribution $p_j(k)$ to account for imperfect size control. A UCB-inspired inverse-count exploration bonus (Auer et al., 2002) encourages under-explored niches, while an annealed size penalty progressively favors smaller niches once quality is secured. Specifically, we have:

$$\mathrm{UCB}(k) \;=\; \underbrace{\sum_{j} p_j(k)\,\big(\widehat{Q}_j - \lambda z_j\big)}_{\text{expected exploitation}} \;+\; \underbrace{\frac{c_e}{\sqrt{n_k + 1}}}_{\text{exploration}}, \qquad k^{*} \;=\; \arg\max_{k}\; \mathrm{UCB}(k), \tag{8}$$

where $p_j(k)$ is the landing distribution of (5) evaluated at arm k's own aimed ask, $\widehat{Q}_j$ is the discounted estimate of the MOQD improvement from a generation landing in niche j, updated as $\widehat{Q}_j \leftarrow (1-\alpha)\widehat{Q}_j + \alpha\Delta$ with update rate α; for an unseen niche, it is initialized to the mean estimate over observed niches. $z_j = (j-1)/(K-1) \in [0, 1]$ is the normalized niche index, imposing greater penalty on larger niches. λ is the size-pressure weight. n_k counts how many generations have targeted niche k, regardless of their realized niches. $c_e > 0$ is the exploration coefficient controlling the strength of the UCB bonus.

Size-Annealing. The penalty weight is a stateless ramp gated on achieved quality,

$$\lambda = \max\big(0,\ \mathrm{hv}^{\mathrm{best}} - H\big), \tag{9}$$

where $\mathrm{hv}^{\mathrm{best}}$ is the best suite hypervolume observed so far on this module, a running maximum over the whole collection. The schedule prioritizes quality before size. Below the quality threshold H, $\lambda = 0$ and no size pressure is applied; above H, λ increases with $\mathrm{hv}^{\mathrm{best}}$, progressively favoring smaller suites as quality improves. This biases the search toward the lexicographic priority of Definition 1: quality is emphasized first, and size pressure is introduced only after sufficient quality has been achieved.

5 EXPERIMENTS

We evaluate four claims in turn: TESEARCH produces stronger and leaner suites than inference-time scaling methods, coding agents, and its own ablations at matched budget (§5.2); its landing model is accurate enough to guide allocation and its online updates converge (§5.3); its acquisition keeps every niche live while still allocating unevenly, and its size-pressure term anneals at the right time at no quality cost (§5.4); and its suites transfer downstream, ruling out reward hacking (§5.5).

5.1 SETUP

Dataset, harness and metrics. We use the 129 distinct Python modules of TestGenEval (Jain et al., 2025), drawn from mature repositories (django, sympy, and others), each with a containerized execution harness (§F); mutants are produced for every module by one generator, `cosmic-ray` (Sixty North, 2015), under a fixed seed, following the same procedure as in TestGenEval (Jain et al., 2025). The mutants are split into a search fold, which scores the search, and a held-out eval fold, which scores what is delivered. For each delivered suite T, we report coverage $c(T)$ (**cov**), mutation score $m(T)$ (**mut**), hypervolume $\mathrm{hv}(T) = c(T) \cdot m(T)$ (**hv**), and suite size $s(T)$ (**size**). Every method runs under an identical call budget $N = 50$ per module. Baseline implementations, hyperparameter calibration, and parameter sensitivity are described in §G.

Baselines. *(i) Search-based software testing*: Pynguin (Lukasczyk & Fraser, 2022), the reference SBST tool for Python, which evolves program inputs against coverage goals using search-based evolutionary algorithms. As its budget is measured in wall clock time rather than call count, we match the budget in wall-clock time: Pynguin is given 6000 s of search per module, the median time a 50-call TESEARCH run takes on this benchmark. *(ii) Inference-time scaling*: iterative refinement (Shinn et al., 2023), which conditions each generation on the last suite and its measured feedback along a single trajectory; best-of-N (Li et al., 2022; Chen et al., 2024), which keeps the best of independent generations; beam search of width 5 over suite revisions; and the context-optimization methods MIPROv2 (Opsahl-Ong et al., 2024) and ACE (Zhang et al., 2026), each adapted to test generation by optimizing the generation context against executed coverage and mutation feedback. GEPA (Agrawal et al., 2026) is a three-objective alternative to our formulation, maintaining a single global Pareto pool over $(c(T), m(T), -s(T))$, where treating $-s$ as an objective admits near-empty suites into the pool as non-dominated before any quality is secured, thus maintaining a front that is wide in size but shallow in quality; OpenEvolve (OpenEvolve contributors, 2025), an open-source implementation of AlphaEvolve (Novikov et al., 2025), evolves suites in a MAP-Elites archive scored by the same feedback. *(iii) Coding agents*: Claude Code (Anthropic, 2025) and Cursor (Anysphere, 2026), instructed to maximize coverage and mutation score while minimizing suite size for the target module under the same call budget, with harness feedback available as tool output. *(iv) Ablations* of TESEARCH: *w/o kernel*, which removes the landing model, so each call asks for the bin center as written and the bandit scores each arm by its own niche rather than by where the call is expected to land, counting pulls where suites land; *w/o UCB*, which removes the acquisition and lets the asks sweep the grid uniformly while keeping the aim; *w/o nearest-parents*, which samples parents at random from the archive fronts; and *w/o* λ, which removes the size pressure by setting $\lambda = 0$. All methods are evaluated on GPT-5.4, Claude Opus 4.8, and Gemini 3.1 Pro, except Claude Code, which uses its native model.

5.2 END-TO-END EFFECTIVENESS

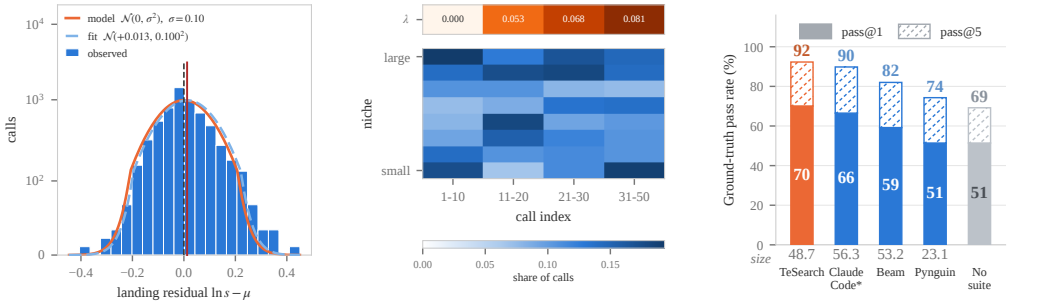
Table 1 shows that TESEARCH achieves the highest delivered hypervolume on all three backbones, exceeding the strongest baseline by 7.0–8.5 hv points while producing substantially smaller suites. Existing inference-time methods either follow a single trajectory, fail to accumulate improvements, or maintain populations without explicitly preserving diversity across suite sizes; context optimizers and coding agents similarly leave suite size uncontrolled. In contrast, TESEARCH explicitly searches the quality–size frontier through size-niched competition. Ablations confirm that the components are complementary: removing any one of them costs hypervolume, but they differ in size. Random parents leave size close to the full system, since allocation is untouched and only each generation's quality drops. Without UCB, the uniform sweep spends as much on large niches as on small ones. Without the kernel, pulls are counted where suites land rather than where they were aimed, so rewards pile up in the larger niches that off-target generations reach and allocation drifts toward them. Removing size annealing inflates suites the most (§5.4). Table 1 compares methods at a fixed call budget. §H evaluates the same runs over token budgets, removing the advantage of TESEARCH's longer per-call prompts, and shows that the ordering persists throughout the search; §I checks that fewer tests also mean less test code and shorter execution.

5.3 LANDING KERNEL ANALYSIS

Figure 2 evaluates the landing model over all TESEARCH generations on GPT-5.4. The size bias δ varies substantially across modules and changes sign, ranging from -0.037 to $+0.107$ in log space, making a fixed correction unsuitable. The online estimate instead adapts to each module, tracking its batch-fitted bias to a median absolute error of 0.022, in line with the LMS steady-

Table 1: **End-to-end comparison** on the 129 TestGenEval modules, means over modules, same call budget for every method (§G.1). **cov**, **mut**, **hv** and **size** describe the delivered suite. Higher is better except **size**: a small suite is only a virtue at matched quality, so size is ranked separately, the smallest per backbone marked in blue. Claude Code runs on its native backbone only; — marks a cell that does not apply. Bold = best per backbone. Pynguin makes no model call, so its budget is matched in wall clock rather than in calls (§5.1); *scored on the 40 modules Pynguin supports (§G.2).

	Method	cov	mut	hv	size								
SBST	Pynguin*	0.464	0.157	0.090	20.4								
		GPT-5.4				Claude Opus 4.8				Gemini 3.1 Pro			
		cov	mut	hv	size	cov	mut	hv	size	cov	mut	hv	size
Inference-time scaling	Iter. refinement	0.798	0.647	0.516	50.8	0.842	0.849	0.716	72.5	0.694	0.764	0.539	56.8
	Best-of-N	0.908	0.758	0.693	47.6	0.953	0.861	0.824	61.5	0.837	0.823	0.691	51.4
	Beam search	0.859	0.905	0.775	44.5	0.971	0.910	0.886	68.8	0.890	0.934	0.838	52.6
	MIPROv2	0.927	0.802	0.746	48.3	0.972	0.891	0.870	63.1	0.859	0.853	0.736	53.1
	GEPA	0.929	0.786	0.732	47.9	0.982	0.911	0.895	67.2	0.845	0.857	0.732	54.7
	ACE	0.914	0.753	0.692	52.4	0.938	0.878	0.827	64.4	0.835	0.826	0.692	55.5
	OpenEvolve	0.963	0.858	0.831	51.5	0.964	0.917	0.885	68.0	0.930	0.853	0.798	56.6
Coding agents	Claude Code	—	—	—	—	0.982	0.908	0.891	51.6	—	—	—	—
	Cursor	0.913	0.762	0.699	44.4	0.967	0.878	0.852	48.2	0.846	0.826	0.701	37.0
Ablations	w/o kernel	0.974	0.884	0.866	40.1	0.988	0.929	0.917	40.8	0.905	0.942	0.855	63.3
	w/o UCB	0.973	0.881	0.861	44.1	0.995	0.964	0.960	34.4	0.906	0.937	0.851	42.3
	w/o near. parents	0.985	0.857	0.847	39.2	0.984	0.949	0.934	33.2	0.922	0.937	0.864	39.4
	w/o λ	0.981	0.880	0.865	58.8	0.991	0.961	0.954	84.6	0.909	0.934	0.852	73.4
	TESEARCH (full)	**0.995**	**0.920**	**0.916**	**38.6**	**0.998**	**0.977**	**0.976**	**29.1**	**0.940**	**0.963**	**0.908**	**36.3**



(a) all landings vs the model

(b) every niche stays active; budget shifts to smaller niches as pressure rises

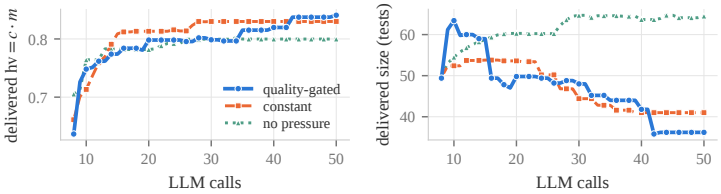
(c) downstream validity: cross-version migration (12 instances)

Figure 2: **The landing model, the acquisition that reads it, and what the suites are worth downstream. (a)** Log-size prediction errors closely follow the Gaussian landing model ($\sigma = 0.10$), with mean residual 0.013 after online bias correction. **(b)** Distribution of target calls across normalized size niches over the search. All size regions continue to receive calls, while the increasing λ shows when size pressure becomes active after quality is secured. **(c)** Downstream validity on cross-version migration over 12 instances: pass rate of the code each condition produced, judged on repository tests, under one coding model, at most five repair rounds and one backbone (GPT-5.4).

state jitter, whose standard deviation $\sqrt{\eta\sigma^2/(2-\eta)}$ is 0.033 at $\eta = 0.2$, and small against the 0.144 spread of δ that a fixed correction would have to absorb. In contrast, the residual spread is relatively stable across modules (median 0.102) and affects only the landing probabilities $p_j(k)$, not the bias-corrected request; we therefore fix $\sigma = 0.10$. Across 6063 generations, the resulting log-size residuals have mean 0.013 and standard deviation 0.100, closely matching the assumed Gaussian model (Figure 2a). The model remains accurate after discretization into niches (§B). Overall, the kernel provides a simple and accurate estimate of where a generation is likely to land, while requiring online adaptation of only the module-specific bias δ.

5.4 Size-Annealing UCB Analysis

Figure 2(b) and Figure 3(a, b) evaluate the two components of the acquisition: UCB allocation across size niches and quality-gated size annealing. The UCB determines *where* to allocate each call, while λ determines *when* to favor smaller suites. We analyze allocation over all 129 modules and compare annealing schedules on a 20-module subset with GPT-5.4 under the same 50-call budget.



(a) anytime quality (b) anytime deliverable size

Figure 3: **The size anneal.** Pressure-schedule comparison on a 20-module subset with GPT-5.4 at the matched call budget. Constant pressure compresses before there is anything worth compressing and saturates lower in quality; no pressure never turns the size curve down at all. Only quality-gating does both, finishing highest on delivered quality and smallest on delivered size.

UCB allocates selectively while preserving exploration. Figure 2(b) groups each module's niches into eight normalized size bands. Although calls are allocated non-uniformly, every band continues to receive calls throughout search, with no band falling below a 0.067 share in any window. This selective allocation is more effective than uniform exploration: removing UCB produces larger suites with lower hypervolume on all three backbones (Table 1).

Quality-gated annealing reduces size and improves quality. We compare three schedules in Figure 3(a, b). With *no pressure* ($\lambda = 0$), suites reach 1.5–$2.9\times$ the size of TESEARCH (Table 1). Once $\widehat{Q}$ decays late in the search, no λz term takes over, so allocation stays in the large niches, where hypervolume is easiest to raise. With *constant pressure* ($\lambda = \lambda_{\max}$), suites do become smaller, but quality saturates earlier and lower, because budget is spent compressing before there is anything worth compressing. Our *quality-gated* schedule avoids both behaviors: λ remains zero until $\mathrm{hv}^{\mathrm{best}}$ exceeds H and then increases with quality, and it finishes above both alternatives on delivered quality while finishing below both on delivered size. That the anneal also buys quality is not incidental: λ steers which niche each call is spent on, removing it does not merely stop suites from shrinking, but also lets the budget concentrate in the large-size niches, which is why *w/o* λ in Table 1 gives up hypervolume as well as compactness. Consistent with this design, the median λ in Figure 2(b) is zero during the first ten calls and increases thereafter as modules reach the quality threshold. Together, UCB preserves exploration across sizes, while quality-gated annealing introduces a preference for smaller suites only after sufficient quality has been achieved.

5.5 Downstream Utility of Optimized Suites

We evaluate whether TESEARCH-optimized suites transfer beyond the setting they were generated in through two downstream tasks: cross-version migration (§5.5.1) and self-evolving agents (§5.5.2). Both consume a suite rather than score it: migration reads it as a guide for the change, and the self-evolving agent runs it as the gate a change must pass.

5.5.1 Cross-Version Migration

Experimental Setup. Coverage and mutation score are optimization proxies, so we test whether the generated suites transfer to a downstream task: cross-version migration. For each of the 12 migration instances that qualify (§J), an LLM migrates the implementation to the next release using one generated suite as guidance; the migrated code is then judged by held-out repository tests that the agent never sees. Using the same backbone (GPT-5.4), we compare five conditions: the delivered suites from TESEARCH, Claude Code*, beam search and Pynguin, and a no-suite baseline. We evaluate two settings. For **pass@1**, the suite is provided as context and the agent performs a single migration without executing it. For **pass@5**, the agent receives up to five repair rounds, executing the suite after each attempt and using test failures as feedback. Here **pass@5** denotes performance after up to five sequential repair attempts, not best-of-five independent sampling. In both settings, the score is the fraction of held-out repository tests the migrated code passes, so these tests serve as the final acceptance oracle.

Results and Analysis. The ordering is TESEARCH, Claude Code*, beam search, then Pynguin, with the unguided agent last (Figure 2c). Migration is partly recoverable without any guide, so the field is compressed, and the best guide is still worth 18.7 points of pass@1 over none. What the guides show is that the transfer follows quality per test rather than test count. Beam search carries fewer tests than Claude Code* and guides worse than it: its suites are the weakest of the three LLM-written conditions on both quality axes (cov/mut $0.832/0.811$ against $0.964/0.909$), and a suite that does

*Claude Code generates its suite with Claude Opus 4.8, since it does not natively support GPT-5.4.

Table 2: Verification signals for self-revisions of mini-swe-agent's harness. div/eq: divergent / equivalent revisions in the module's pool. DDR: divergent revisions rejected (↑); FAR: equivalent revisions rejected (↓); s: mean wall clock per decision.

			Repository tests			TESEARCH		
Module	Role in the harness	div/eq	DDR	FAR	time (s)	DDR	FAR	time (s)
`agents/default`	agent loop: model calls, actions, limits	17/41	0.529	0.000	15.7	**0.588**	0.049	3.0
`environments/local`	runs bash commands, captures output	25/31	0.720	0.000	4.9	**0.920**	0.065	3.3
`actions_toolcall`	parses tool calls; renders observations	29/34	0.276	0.000	3.0	**0.552**	0.029	3.1
`actions_text`	parses text actions; renders observations	25/35	0.040	0.000	7.7	**0.560**	0.029	2.9
`cache_control`	places prompt-cache breakpoints	29/26	0.172	0.000	2.9	**0.414**	0.038	3.0
`content_string`	flattens messages to text for display	35/25	0.057	0.000	2.9	**0.600**	0.040	2.9
`litellm_model`	model client: API call, retry, cost	38/25	0.026	0.000	7.1	**0.421**	0.080	6.7
`serialize`	merges layered config dictionaries	19/36	0.579	0.000	2.9	**0.947**	0.000	3.1
Pooled		217/253	0.253	0.000	5.9	**0.599**	0.040	3.5

not pin behavior down is not rescued by the tests it does contain. Pynguin makes the same point from the other end. It carries the fewest tests, and its pass@1 is identical to having no suite at all, so as context it contributes nothing an agent can read; it earns its only margin at pass@5, where the suite is executed rather than read. TESEARCH is the smallest of the LLM-written suites and also the strongest on both axes (0.996/0.934), and it leads Claude Code* by 3.6 points of pass@1 while carrying 13% fewer tests. Leanness therefore costs nothing downstream: the same behavior is pinned by less context for the agent to read and fewer failures for it to chase. The gains persist under held-out repository tests, indicating that the optimized metrics translate to downstream utility rather than reward hacking.

5.5.2 Enhancing Self-Evolving Agents with TESEARCH

Experimental Setup. A self-evolving agent revises its own harness and keeps a revision only if it passes a verification gate (Lou et al., 2026; Wang et al., 2026). The gate must reject revisions that change behavior without blocking those that preserve it: an accepted behavior change carries into every later self-modification, and a rejected sound improvement stalls evolution. We study this decision on mini-swe-agent (SWE-agent team, 2025), an open-source variant of SWE-agent (Yang et al., 2024b). For each of eight harness modules, GPT-5.4 receives a revision intent drawn from common self-improvement goals, either behavior-preserving or behavior-changing, and implements it as a revised module, yielding 64 candidates per module (§K.1). The revision is further executed and verified by human annotators (§K.2), leaving the 470 valid revisions reported in Table 2. A revision is labeled *divergent* if its intent is behavior-changing and *equivalent* otherwise. We compare the suite generated by TESEARCH from the original module, and the module's repository tests on the revisions. We report the divergent-revision detection rate (**DDR**), the equivalent-revision false-alarm rate (**FAR**), and the verification cost per decision (§K.3).

Results and Analysis. TESEARCH catches more than twice as many divergent revisions as the repository tests, and more on every module (Table 2). The two gates constrain different things. Repository tests check that a module performs its intended function, whereas a mutation-optimized suite must also pin down the details a mutant perturbs, such as a separator, a default value, or which exception surfaces. The gap is therefore widest on the modules that parse model outputs or render what the model reads, where a revision can keep the nominal function intact while changing what the model observes. This tighter constraint costs little: TESEARCH rejects 10 of the 253 equivalent revisions, whereas repository tests reject none. Decisions are also faster on average, with the savings concentrated on the modules whose repository tests are slowest to run, and neither gate calls a model at decision time. Optimizing a suite for both effectiveness and size therefore yields an acceptance signal for self-evolving agents that is both stronger and cheaper.

6 Conclusion

We introduced TESEARCH, a quality–diversity evolutionary search for *size-optimized unit test generation*. Our key formulation treats suite size as a behavioral descriptor rather than trading it directly against test quality, enabling quality to be optimized across the size spectrum. Because LLMs only approximately follow size instructions, TESEARCH models the mapping from requested to realized size with an online transition kernel and incorporates this uncertainty into a size-annealing UCB. Across three frontier LLMs, TESEARCH produces stronger and smaller suites than existing inference-time search methods and coding agents, with downstream gains in cross-version migration and verification for self-evolving agents. More broadly, TESEARCH shows how quality–diversity search can optimize structured LLM outputs whose behavioral descriptors are desired but only imperfectly controllable.

AI USE STATEMENT

In our work, we used large language models (LLMs) for two purposes: (1) as a general writing assistant to check the grammar of the manuscript for readability, and (2) as the model component of our experiments, where LLMs serve as the test generator inside TESEARCH and the baselines, as the coding agents we compare against, and as the agents that perform the cross-version migrations and the self-revisions in §5.5. Every experimental outcome is measured by executing code, through coverage, mutation score and held-out repository tests, or by human annotators, and no LLM is used as an evaluation judge. All research ideas, methodological contributions, and conclusions are solely those of the authors, who take full responsibility for the content of this work.

REPRODUCIBILITY STATEMENT

The full implementation of TESEARCH is included as anonymized supplementary material. The appendix documents what the code implements. The size-niched grid is defined in §A and the complete search loop is given in §C; every prompt is reproduced verbatim in §E; the benchmark, the per-module containers and the fixed-seed mutant generation are described in §F; and §G details each baseline implementation, every hyperparameter with the source of its value, and how the two calibrated constants were chosen. The cross-version migration and self-evolving agent experiments, including dataset construction and the human-verified labels, are specified in §J and §K.

ETHICS STATEMENT

Our work aims to make the verification of LLM-based coding agents both cheaper and stronger by developing TESEARCH, which generates compact unit-test suites that can be executed repeatedly inside agentic loops. We believe our research has positive broader impacts, as fast and effective acceptance checks make it more practical to catch faulty changes before coding agents and self-evolving agents adopt them. However, we also acknowledge potential risks. A generated suite checks behavioral equivalence, namely that a revision behaves as the original code does, and not behavioral safety: if the original code is itself unsafe, a revision that faithfully preserves its behavior passes the suite and is equally unsafe, so treating a passing suite as a safety guarantee could lead practitioners to over-trust agent revisions. Our experiments also execute model-generated code. To mitigate these risks, we run all model-generated code, including the suites, the coding-agent sessions and the agent self-revisions, in isolated per-module containers with web access withheld from the coding agents, so that nothing this code does reaches the host or external systems, and we recommend that generated suites complement rather than replace human review wherever an agent's changes carry real consequences. The benchmark modules, the migration instances and mini-swe-agent are publicly available open-source Python code, and no dataset in this work contains personal data. The only human involvement is the labeling of agent self-revisions in §K.2, where annotators judged whether a code change realizes its stated intent; no information about the annotators is used beyond the background reported there. All authors have read and adhere to the ICLR Code of Ethics, and we encourage the community to use our work responsibly to advance reliable and trustworthy coding agents.

REFERENCES

Rishabh Agarwal, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G. Bellemare. Deep reinforcement learning at the edge of the statistical precipice. In *NeurIPS*, 2021.

Lakshya A. Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J. Ryan, Meng Jiang, et al. GEPA: Reflective prompt evolution can outperform reinforcement learning. In *ICLR*, 2026.

Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at Meta. In *FSE Companion*, 2024.

Anthropic. Claude Code: an agentic coding tool, 2025. URL `https://claude.com/claude-code`.

Anysphere. Cursor: AI coding agent for building ambitious software, 2026. URL https://cursor.com.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 2002.

Chenjia Bai, Yang Zhang, Shuang Qiu, Qiaosheng Zhang, Kang Xu, and Xuelong Li. Online preference alignment for language models via count-based exploration. In *ICLR*, 2025.

Guoxin Chen, Minpeng Liao, Chengxi Li, and Kai Fan. AlphaMath almost zero: Process supervision without process. In *NeurIPS*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Antoine Cully and Yiannis Demiris. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 2018.

Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 2002.

DeepSeek-AI. DeepSeek Harness: Everything is a plugin, 2026. URL https://github.com/deepseek-ai/deepseek-harness.

Xidong Feng, Bo Liu, Yan Song, Haotian Fu, Ziyu Wan, Girish A. Koushik, Zhiyuan Hu, Mengyue Yang, Ying Wen, and Jun Wang. Natural language reinforcement learning. *arXiv preprint arXiv:2411.14251*, 2025.

Matthew Fontaine and Stefanos Nikolaidis. Differentiable quality diversity. In *NeurIPS*, 2021.

Gordon Fraser and Andrea Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *ESEC/FSE*, 2011.

David E. Goldberg and Jon Richardson. Genetic algorithms with sharing for multimodal function optimization. In *ICGA*, 1987.

Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 1993.

Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE*, 2014.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. TestGenEval: A real world unit test generation and test completion benchmark. In *ICLR*, 2025.

Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 2011.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution. In *ICLR*, 2026.

Joel Lehman and Kenneth O. Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation*, 2011a.

Joel Lehman and Kenneth O. Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *GECCO*, 2011b.

Jiaming Li, Lei Zhang, Yunshui Li, Ziqiang Liu, Yuelin Bai, Run Luo, Longze Chen, and Min Yang. Ruler: A model-agnostic method to control generated length for large language models. In *Findings of EMNLP*, 2024.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with AlphaCode. *Science*, 2022.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 2024.

Xinghua Lou, Miguel Lázaro-Gredilla, Antoine Dedieu, Carter Wendelken, Wolfgang Lehrach, and Kevin P. Murphy. AutoHarness: improving LLM agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for Python. In *ICSE Companion*, 2022.

Charles Masson, Jee E. Rim, and Homin K. Lee. DDSketch: A fast and fully-mergeable quantile sketch with relative-error guarantees. *Proceedings of the VLDB Endowment*, 2019.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA*, 2009.

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

OpenAI. Introducing Codex, 2025. URL https://openai.com/index/introducing-codex/.

OpenEvolve contributors. OpenEvolve: An open-source implementation of AlphaEvolve, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Krista Opsahl-Ong, Michael J. Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *EMNLP*, 2024.

Thomas Pierrot, Guillaume Richard, Karim Beguir, and Antoine Cully. Multi-objective quality diversity optimization. In *GECCO*, 2022.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 2024.

Gregg Rothermel, Mary Jean Harrold, Jeffery von Ronne, and Christie Hong. Empirical studies of test-suite reduction. *Software Testing, Verification and Reliability*, 2002.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

Sixty North. Cosmic ray: mutation testing for Python, 2015. URL https://github.com/sixty-north/cosmic-ray.

SWE-agent team. mini-SWE-agent: the 100-line AI agent that solves GitHub issues, 2025. URL https://github.com/SWE-agent/mini-swe-agent.

Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. Refactoring test code. Technical Report SEN-R0119, CWI, 2001.

Tim van Erven and Wouter M. Koolen. Metagrad: Multiple learning rates in online learning. In *NeurIPS*, 2016.

Xiaoxuan Wang, Haixin Wang, Alexander Taylor, Jason Cong, Yizhou Sun, and Wei Wang. HarnessBridge: learnable bidirectional controller for LLM agent harness. *arXiv preprint arXiv:2606.12882*, 2026.

Bernard Widrow and Marcian E. Hoff. Adaptive switching circuits. In *Neurocomputing: Foundations of Research*. MIT Press, 1988.

Meiman Xiao, Ante Wang, Qingguo Hu, Zhongjian Miao, Huangjun Shen, Longyue Wang, Weihua Luo, and Jinsong Su. Can LLMs track their output length? a dynamic feedback mechanism for precise length regulation. *arXiv preprint arXiv:2601.01768*, 2026.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024a.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024b.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 2012.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, et al. Agentic context engineering: Evolving contexts for self-improving language models. In *ICLR*, 2026.

Wei Zhang, Zhenhong Zhou, Kun Wang, Junfeng Fang, Rongwu Xu, Yuanhe Zhang, Rui Wang, Ge Zhang, Xinfeng Li, Li Sun, Lingjuan Lyu, Yang Liu, and Sen Su. LIFEBench: Evaluating length instruction following in large language models. In *NeurIPS Datasets and Benchmarks Track*, 2025.

Zihao Zhao, Eric Wallace, Shi Feng, Dan Klein, and Sameer Singh. Calibrate before use: Improving few-shot performance of language models. In *ICML*, 2021.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

Eckart Zitzler and Lothar Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Transactions on Evolutionary Computation*, 1999.

APPENDIX CONTENTS

A The size-niched grid **14**

B Landing kernel analysis **15**

C The search loop **15**

D Parent selection and context assembly **15**

E Prompt construction **17**

F Benchmark and harness **19**

G Experimental setup **20**
G.1 Baseline implementations 20
G.2 Pynguin 21
G.3 Hyperparameters 21

H Anytime behavior under a token budget **24**

I Suite size as a proxy for verification cost **25**

J Cross-version migration **26**
J.1 Dataset construction 26

K Self-evolving agent verification **27**
K.1 The revision pool 27
K.2 Label construction and human verification 28
K.3 Metrics 29

A THE SIZE-NICHED GRID

Suite size is the archive's only descriptor, and the noise on it is multiplicative: asking for twice as many tests does not add a fixed error, it scales the error, which is why §4.1 models landing as log-normal. The grid is therefore a partition of $\ln s$ built once, immediately after the seed phase. The grid is centered on an *anchor* $\bar{s}$, the size the backbone writes at when no size is named, estimated as the geometric mean of the seed suites' sizes. Writing $T_1, \ldots, T_n$ for the n seed suites that measured,

$$\bar{s} \;=\; \exp\Big(\tfrac{1}{n}\sum\nolimits_{i=1}^{n} \ln s(T_i)\Big). \tag{10}$$

The mean is taken on the log scale because the noise is multiplicative, and the seeds are unhinted precisely so that the anchor measures the model rather than our own prompt.

Niches are geometric, with equal width on the log scale. This is the spacing used whenever values within a constant factor of each other are interchangeable, as in MetaGrad's exponentially spaced grid of learning rates (van Erven & Koolen, 2016), and whenever error is relative rather than absolute, as in DDSketch, whose bucket ratio is set by the relative accuracy it guarantees (Masson et al., 2019). Under multiplicative noise, both hold for suite sizes. Consecutive niches differ by the ratio $\rho = e^{2\sigma} = 1.22$, where $\sigma = 0.10$ is the landing spread of the transition kernel (§4.1), so each niche spans $\pm\sigma$ around its center on the log scale. This width is intended: since §4.1 models landing noise as log-normal, an aimed draw lands in the niche it was aimed at with probability 68% under the model, and 76% of the measured residuals do (§B).

The grid is built per module over the range $[s_{\min},\ \bar{s}\rho^{\gamma}]$, and both ends follow conventions of software testing (van Deursen et al., 2001; Namin & Andrews, 2009). The lower end is set by the module: $s_{\min}$ is the number of public functions and methods of the module under test, counted statically from its AST. A suite with fewer tests than that must check several of them in one test method, which is the *Eager Test* smell: such a test is hard to read, hard to use as documentation of the behavior it checks, and makes tests more dependent on each other, and the prescribed remedy is to split it until each test method exercises a single method of the unit (van Deursen et al., 2001). The floor also guards against reward hacking, in which the LLM packs all its checks into a single test function to game the size reward. The upper end is relative to the anchor, γ niches above it, and so scales with the size of the module. In software testing, the effectiveness of a suite grows with its size (Inozemtseva &

Holmes, 2014), but only logarithmically (Namin & Andrews, 2009). Beyond a minimal suite, one from which no test can be removed without lowering its mutation score, further tests add nothing but code smells such as *Lazy Test* and *Test Code Duplication* (van Deursen et al., 2001). To keep suite size from growing without bound, and to avoid spending calls on unreasonably large suites, we set $\gamma = 4$. The boundaries that fall inside the range, sorted and with duplicates dropped, are $b_1 < \cdots < b_{K-1}$. A suite outside the range is filed rather than dropped, into one of two open overflow niches, $[0, b_1)$ and $[b_{K-1}, \infty)$; the bandit never aims at them, and a call that lands in one is still charged to the arm it aimed at.

B LANDING KERNEL ANALYSIS

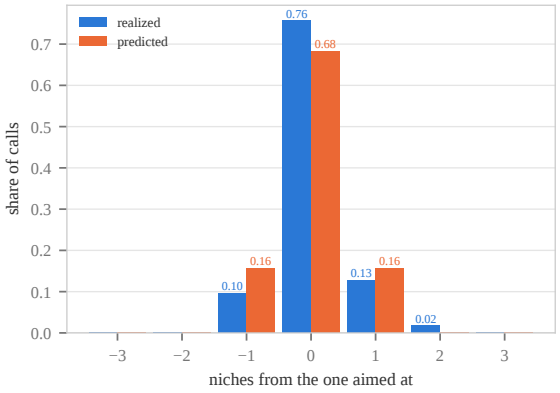


Figure 4: **The kernel predicts where draws land.** Predicted versus observed landing probabilities relative to the target niche, over the same 6063 residuals as Figure 2(a), binned on the grid of §A, whose niches are $\ln \rho = 2\sigma$ wide. The kernel predicts 68% on-target landings versus 76% observed, with the misses almost all in adjacent niches.

§5.3 checks the landing model on the continuous log scale. The UCB acquisition, however, reads it only after discretization into size niches, and the model remains accurate there. Each niche is $\ln \rho = 2\sigma$ wide (§A), spanning its center $\pm\sigma$, and at that width a centered Gaussian predicts an on-target landing rate of 68%, against 76% observed, with the misses almost all in adjacent niches (Figure 4). The observed rate exceeds the predicted one because the residual is leptokurtic (excess kurtosis $+2.5$): the Gaussian is therefore not an exact distributional fit, nor is one required. It serves as an operational model for the niche-level landing probabilities used by the acquisition. At this granularity, it predicts 68% on-target landings versus 76% observed, with almost all misses in adjacent niches; the discrepancy is conservative for an acquisition that spreads credit over neighboring niches.

C THE SEARCH LOOP

Algorithm 1 states the whole method.

D PARENT SELECTION AND CONTEXT ASSEMBLY

Given the selected niche k^*, TESEARCH selects two parents from all Pareto-front suites in the archive, rather than only from the target niche. Specifically, one parent is taken from each side of the target size: on each side, suites are ranked by their log-size distance to the target,

$$|\ln s(T) - \ln \tau_{k^*}|\,,$$

with hypervolume breaking ties, and the nearest suite is selected. Using the global archive allows generation even when the target niche is empty, while size proximity provides examples at a scale close to the requested suite.

Algorithm 1 TESEARCH on one module. LLM is the frozen backbone; MEASURE executes a suite in the module's container and returns (c, m) on its passing subset together with its written size s.

Input: module M; call budget N; seeds n_{seed}; hyperparameters (Table 7)
Output: one delivered suite

Seed phase — the grid does not exist yet, so nothing can be discarded on it.
for $i = 1, \ldots, n_{\text{seed}}$ **do**
  $T_i \leftarrow \text{LLM}(\text{SEED}, M)$ *// unhinted: no count is named*
  MEASURE(T_i); file T_i provisionally, dominance off
end for
$\bar{s} \leftarrow$ Eq. (10) on the measured seed sizes; $s_{\min} \leftarrow$ public functions and methods of M
derive $b_0 \ldots b_K$ over $[s_{\min}, \bar{s}\rho^{\gamma}]$ (§A); re-file the archive under them
$\delta \leftarrow 0$; $n_k \leftarrow 0$ for all k; $\text{hv}^{\text{best}} \leftarrow \max_i \text{hv}(T_i)$
replay the seed landings into $\widehat{Q}$ *// a seed asked for no size Search.*
while fewer than N calls have been spent **do**
  if the archive is empty **then** *// only after every seed failed to measure*
    regenerate unhinted, measure, file; **continue**
  end if
  for $k = 2, \ldots, K-1$ **do** *// the overflow niches* 1 *and* K *are never aimed at*
    $P_k \leftarrow$ the front member nearest τ_k in log size on each side of τ_k *// §D*
    $a_k \leftarrow \lfloor \exp(\ln \tau_k - \delta) \rceil$ *// the aim, Eq. (4)*
    $p_j(k) \leftarrow$ Eq. (5) at a_k, for every j *// the kernel, run forward*
    $S_k \leftarrow \sum_j p_j(k)\big(\widehat{Q}_j - \lambda z_j\big) + c_e/\sqrt{n_k + 1}$ *// Eq. (8)*
  end for
  $k^* \leftarrow \arg\max_{2 \le k \le K-1} S_k$ *// deterministic; the bonus already schedules variety*
  mode $\leftarrow$ COMPRESS if τ_{k^*} lies below the last parent of P_{k^*}, else EXPAND *// Table 4*
  $T \leftarrow \text{LLM}\big(\text{BREED}[\textit{mode}],\ M,\ P_{k^*},\ a_{k^*}\big)$
  $n_{k^*} \leftarrow n_{k^*} + 1$ *// the* ask *is charged, measured or not*
  if MEASURE(T) succeeds **then**
    file T in the niche of its *written* size, $j \leftarrow \text{niche}(s(T))$
    $\Delta \leftarrow$ increment of Eq. (7), clipped to $[0, 1]$ *// the reward*
    $\delta \leftarrow (1 - \eta)\,\delta + \eta\big(\ln s(T) - \ln a_{k^*}\big)$ *// Eq. (6)*
    $\widehat{Q}_j \leftarrow \widehat{Q}_j + \alpha\big(\Delta - \widehat{Q}_j\big)$ *// credited where it* landed
    $\text{hv}^{\text{best}} \leftarrow \max\big(\text{hv}^{\text{best}}, \text{hv}(T)\big)$; recompute λ by Eq. (9)
  end if
end while
return the smallest-written suite among those of maximal hv *// Definition 1*

The generation context consists of the module under test, execution feedback, the two selected parents, and the requested size. Execution feedback highlights uncovered lines and surviving mutants. The parents are ordered from weaker to stronger, placing the stronger example closest to the generation request. Finally, the LLM is instructed to generate a suite of the requested size by compressing or extending the last parent, according to whether the target lies below or above it. §E provides the full prompt.

Parent selection. Parents are selected by size proximity from the two sides of the target: the Pareto-front suite nearest τ_{k^*} in log size among those smaller than the target, and the nearest among those larger. Because the grid is geometric, the niche index is log size up to a constant, so each side is a one-dimensional nearest-neighbor lookup. The two parents therefore always differ in size, and their relation in Table 3 follows from their hypervolumes: if the smaller parent has no lower hypervolume, it Pareto-dominates the larger one; otherwise the larger one lexicographically dominates it. When one side has no suite, the parents come from the other side alone; in that edge case the call must compress or expand the suite toward the target while raising its hypervolume. The pool is global rather than niche-local for a structural reason: the acquisition frequently selects a niche no suite has reached yet, and a draw restricted to that niche's own front would leave exactly those targets unbreedable. The *w/o nearest-parents* ablation replaces the ranking with a uniform draw

over the same pool and changes nothing else — the candidates are still front members, so nothing dominated is ever shown, and the ordering rule below still applies.

Ordering and annotation. The two parents and the target are arranged ascending under the order of Definition 1 — by hypervolume first, and by smaller size only when hypervolumes are equal — with the target always last, immediately before the continuation, where recency weighs the nearest example most (Zhao et al., 2021; Liu et al., 2024; Yang et al., 2024a). With one parent on each side of the target size, the arrangement takes one of two mirror forms (Table 4). If the larger parent L has the higher hypervolume, the order is S, L, T: L lexicographically dominates S, and the target T must Pareto-dominate L, compressing it to the target size at no lower hypervolume. Otherwise the order is L, S, T: S Pareto-dominates L, and T must lexicographically dominate S, expanding it to the target size at strictly higher hypervolume. The asymmetry is deliberate: a smaller suite at equal hypervolume is already an improvement under Definition 1, and the only one left once hypervolume saturates, whereas a larger suite is worth its extra tests only if it raises hypervolume. The second parent carries a one-line note comparing it with the first, in one of the two forms of Table 3; the note states the measured change in both hypervolume and size, so it is always a measured claim rather than an inferred narrative.

Table 3: The comparison note attached to the second parent. Together with the target, which always comes last, the ordered parents form a gradient along the order on $(\mathrm{hv}, \mathrm{size})$, so the strongest example sits nearest the output position, where the model's recency bias weighs it most (Zhao et al., 2021; Liu et al., 2024; Yang et al., 2024a). Because the parents lie on opposite sides of the target size and are ordered first, the later of the two is never the weaker, so the relation reduces to two cases, and the note reports both changes with their numbers.

Type	Condition on (prev, curr)	Emitted note
Pareto dominance	no lower hypervolume and smaller size	`perfect improved version of prev (hypervolume grow 0.852->0.916, size decline 24->13)`
lexicographic dominance	higher hypervolume and larger size	`improved version of prev (hypervolume grow ..., size grow ...)`

Table 4: What the target T must achieve against the parent placed next to it. S and L are the parents below and above the target size. The requirement is asymmetric in hypervolume because the step toward T changes size in opposite directions.

Case (order)	T against its neighbor	Hypervolume of T	Emitted note
compress (S, L, T)	smaller than L	$\mathrm{hv}(T) \geq \mathrm{hv}(L)$	`perfect improved version of prev (hypervolume grow or hold from <hv(L)>, size decline <size(L)>-><n>)`
expand (L, S, T)	larger than S	$\mathrm{hv}(T) > \mathrm{hv}(S)$	`improved version of prev (hypervolume grow from <hv(S)>, size grow <size(S)>-><n>)`

E PROMPT CONSTRUCTION

With the two parents chosen and ordered (§D), three further decisions turn them into a prompt: which of the two breeding framings the call is given, what execution signal is written into the module source, and how much of each piece is shown.

Feedback signal. The module source shown in a breeding prompt is annotated with what the *strongest* parent still misses, written at the end of the exact line it describes rather than as a separate

list of line numbers: `# <-- NOT COVERED YET` on a line none of its passing tests execute, and `# <-- SURVIVING MUTANT: <op>` on a line where that mutation survives its assertions. Failing tests inside an example suite are marked the same way, at the end of the (possibly multi-line) `def` header, as `# <-- FAILING: <exception>`. Adjacency is the point: models align a mark sitting next to the code far more reliably than a position given in a distant block of text. Coverage only ever flags executable statement lines, so a trailing comment is syntactically safe wherever a mark can land.

Prompts, verbatim. Below are the three system prompts and the user prompt skeleton, exactly as the implementation emits them; a bracketed `[...]` marks text repeated verbatim from the prompt above it, and `<...>` marks a slot filled at run time. Throughout this appendix, system prompts are marked in blue and user prompts in gray.

System Prompt — SEED

```
You write ONE complete, self-contained Python test suite (pytest style) for the target
    module. Output ONLY the suite source - no prose, no fences. The suite must include its
     own fixture scaffolding at module level: import the SUT, bootstrap the runtime, stub
    collaborators, build inputs - then the tests. EVERY test must pass against the module
    as given. Compute expected values yourself by reading the module source. Prefer many
    small, strongly-asserting tests spread across ALL functions/classes of the module.
```

System Prompt — BREED, compression mode

```
You write ONE complete, self-contained Python test suite (pytest style) for the target
    module - a GLOBAL, whole-suite generation, not targeted at any one function or region.
     Output ONLY the suite source - no prose, no fences. The suite must include its own
    fixture scaffolding at module level: import the SUT, bootstrap the runtime, stub
    collaborators, build inputs - then the tests. EVERY test must pass against the module
    as given. Compute expected values yourself by reading the module source. You will be
    shown EXISTING versions of this suite, ordered from weakest to strongest by
    hypervolume (coverage x mutation-kill), the smaller one ranking stronger when
    hypervolumes are equal; the LAST one is the strongest shown. Each is labeled with its
    own size/coverage/mutation-kill numbers, and the second carries a short, factual note
    comparing it to the version immediately before it. Some tests in the versions you are
    shown FAIL against the module as given; each is marked `# <-- FAILING: <exception>` at
     the end of its `def` line, and the numbers in that version's header were measured on
    its PASSING tests only. Your task is CONSOLIDATION, not writing from scratch: the
    target size is BELOW the last version. Rewrite that version DOWN to the target size
    with a hypervolume at least as high as its own, and higher if you can: merge related
    assertions into fewer, denser tests; share setup; fold parametrizable cases together
    with pytest.mark.parametrize; drop only what is truly redundant. Do NOT drop that
    version's strong, exact-value assertions - they are what kills the mutants. Repair a
    failing test if its intent is worth keeping at the target size; otherwise drop it - it
     is contributing nothing to the numbers today.
```

System Prompt — BREED, extension mode

```
[...] Your task is EXTENSION, not writing from scratch: the target size is ABOVE the last
    version. Grow that version UP to the target size, and make the added tests raise its
    hypervolume: a larger suite whose hypervolume does not rise is worse than the version
    you started from. KEEP that version's tests and their strong, exact-value assertions (
    verbatim or lightly refactored), and ADD new, strongly-asserting tests that execute
    the `# <-- NOT COVERED YET` lines and assert exact values/branches around each `# <--
    SURVIVING MUTANT` so the mutation would fail. Do not pad with redundant tests - every
    added test should close an annotated gap. Failing tests carry real testing intent -
    REPAIR them by reading the module source and fixing the expected value, rather than
    deleting them, unless the same behaviour is already checked elsewhere.
```

The user prompt is the three pieces and the ask, in order: the module with the residual signal marked inline, the count, then the ordered parents, strongest last.

User Prompt — BREED (<...> are substituted at run time)

```
[TARGET MODULE] <dotted.module.path>

[MODULE SOURCE - annotated with what the LAST (best) suite below still misses: `# <-- NOT
    COVERED YET` marks lines none of its tests execute; `# <-- SURVIVING MUTANT: <op>`
    marks lines where that mutation survives its assertions. Cover those lines, and assert
     exact values/branches around each surviving mutant so the mutation would fail]
<module source, in full, with inline residue marks>
```

```
[TARGET SIZE] exactly 18 test functions. This is a HARD REQUIREMENT: the suite MUST contain
      exactly 18 functions whose name starts with `test_`. Count them before you finish and
      adjust until the count is 18

[test_suite_v0 - size=24 green=22 pass_rate=0.917 coverage=0.947 mutation=0.900 loc=476]
(2 of its 24 tests FAIL - each marked inline below; the coverage/mutation above come from
     the 22 that pass)
<weaker parent, in full>

[test_suite_v1 - size=13 green=13 pass_rate=1.000 coverage=0.982 mutation=0.933 loc=273]
(perfect improved version of prev (hypervolume grow 0.852->0.916, size decline 24->13))
<stronger parent, in full>

Write test_suite_v2 now: the next, improved complete suite. Complete file, no prose, no
     fences.
```

An arm whose niche the archive cannot supply parents for at all — only possible before the archive has any occupant — falls back to the seed prompt with the size instruction appended.

F BENCHMARK AND HARNESS

Dataset. TestGenEval-lite (Jain et al., 2025) pairs each instance with a source module, its repository's own test file for that module, and a containerized testbed pinned to the right release. We use the `lite` split, whose 160 instances cover 129 distinct modules — an instance is a (module, commit) pair, so the same file recurs at several releases, and a search run is defined by the module rather than by the issue. All reported means are over those 129 modules.

Table 5: The 129 modules, by repository. Module size is measured in source lines of the file under test.

Repository	Modules	Repository	Modules
django/django	48	matplotlib/matplotlib	7
sympy/sympy	31	astropy/astropy	3
scikit-learn/scikit-learn	20	pydata/xarray	3
pytest-dev/pytest	10	seaborn, sphinx, pylint	2 each
		pallets/flask	1
Module size: median 571 lines, mean 870, range 25–3876.			

Execution environment. The search — scheduler, archive, prompt assembly — runs on the host and produces only text. All execution happens in a resident per-module container built from that instance's testbed image, started once and reused by `docker exec` for the whole run. The host never imports the repository under test and the LLM never runs inside the container. Two image families occur in the benchmark and are probed for at startup rather than assumed: conda-based testbeds, where the repository lives under the service user's home and commands are wrapped in `conda run`, and pyenv-based ones, where it lives under `/opt` and the shims resolve the interpreter. Some django testbeds run Python 3.6, so the in-container measurement helper is transpiled to 3.6-safe source before it is copied in; without that, a syntax error in our own helper would present as a stuck module.

Measuring a suite. One measurement produces two snapshots of the same object. The *raw* snapshot is the suite exactly as the model wrote it: its size — the count of `test_` functions, which is the archive's descriptor — and its pass rate. The *green* snapshot is its passing subset, and it is what the quality axes are computed on. The split is necessary rather than fastidious: a test that fails on the unmutated module trivially "kills" every mutant, so mutation score is only meaningful on green tests, and coverage has to be re-probed rather than reused, because a failing test executes part of the module before it raises and raw coverage therefore overstates what the passing subset reaches. A suite with no tests at all, or none that pass, is not filed: it has no niche to occupy and no fitness to defend, and would sit in a sparse niche as a zero-hypervolume incumbent shown to the model as that niche's best.

Mutants. Mutants are generated once per module at container build time by `cosmic-ray` under a fixed seed, following TestGenEval's procedure (Jain et al., 2025; Jia & Harman, 2011). Enumeration is round-robin *across operator families* rather than across operators, because the tool lists dozens of binary-arithmetic operators first and a raw round-robin would fill the cap from one family before reaching the rest; it stops once the two folds are full, which bounds the cost on large modules. The pool is then split into two disjoint folds. The *search fold* is 60 mutants, picked by a region-then-family stratification — a minimum of one per function that has mutants, the remainder allocated proportionally by largest remainder, and within a region the candidate whose family is most under-represented against the pool's family proportions. The remainder is the held-out *eval fold*. The search fold scores the search; the eval fold scores what is delivered, so the number reported for a suite was never optimized against. The eval-fold mutant sources never leave the container, which makes the separation structural rather than procedural. Table 6 lists the operator families.

Table 6: Mutation operator families, as the stratification groups them.

Family	Operators
binary	`ReplaceBinaryOperator*`
comparison	`ReplaceComparisonOperator*`
unary	`ReplaceUnaryOperator*`
number	`NumberReplacer`
logic	`AddNot`, `ReplaceAndWithOr`, `ReplaceOrWithAnd`
boolean	`ReplaceTrueWithFalse`, `ReplaceFalseWithTrue`
loopflow	`ReplaceBreakWithContinue`, `ReplaceContinueWithBreak`, `ZeroIterationForLoop`
decorator	`RemoveDecorator`
exception	`ExceptionReplacer`
variable	`VariableInserter`, `VariableReplacer`

G Experimental setup

This appendix collects what is needed to rerun and to trust the comparison of Table 1, in four steps: how each baseline is implemented and what had to change to fit it to this harness (§G.1 and §G.2); every hyperparameter, with where its value comes from (§G.3); how the two constants that are calibrated rather than derived were chosen (§G.3.1); and how much the result depends on that choice (§G.3.2). Unless stated otherwise, every run uses the values of Table 7, and every baseline keeps its own published defaults, scaled to the budget only where a stage count would not fit.

G.1 Baseline implementations

Every baseline is measured by the same harness: the same containerized measurement, the same budget ledger, and the same delivery rule applied to the suites its budget produced. A baseline therefore replaces the search policy while the measurement stays fixed. Three rules keep the comparison fair, and they decide what the numbers mean. First, one budget unit is one LLM call for every method, meta calls included: a method that spends a third of its budget on reflection has a third fewer suites to show for it, which is a property of the method rather than an artifact of the measurement. Pynguin, which makes no model call, is the one exception and is matched in wall clock instead (§5.1). Second, every generated suite is measured and recorded, including candidates the method itself discarded, such as a GEPA rollout that failed its acceptance filter or a beam child that lost its slot, so that MOQD and the delivered suite are computed over everything the budget actually produced. Third, the LLM-based baselines generate with TESEARCH's prompt unchanged: the same whole-suite framing and the same inline feedback signal, never truncated. What such a baseline may vary is only what its own paper varies, namely which prior suites are shown or what instruction sits in the system slot. Every baseline is given an explicit size instruction and is told to optimize coverage, mutation score and suite size jointly.

Inference-time scaling. *Iterative refinement* keeps one trajectory: call $t{+}1$ sees suite t with its failures marked and the module annotated with what t still misses, and the chain is kept through a

regression, which is what makes it a trajectory rather than a hill-climb. *Best-of-N* draws N independent generations from one prompt with no feedback crossing calls; it is the control for how much of a gain is simply repeated draws. *Beam search* keeps 5 suites; a layer revises every member once and then keeps the best 5 of parents and children together, so a layer can never make the beam worse, and members are ranked by the delivery order (hypervolume, ties to smaller and shorter). *MIPROv2* (Opsahl-Ong et al., 2024) runs its three stages against the same budget: 6 bootstrap generations collect demonstrations, 8 candidate instructions are proposed from them in 2 calls, and the remaining calls are trials over the instruction $\times$ demonstration-set grid driven by a categorical TPE, each trial scored by its suite's hypervolume. *GEPA* (Agrawal et al., 2026) spends two calls per round, a reflection that reads execution feedback and rewrites the instruction and a rollout that runs it, with 20% of rounds merging two Pareto-pool members instead. *ACE* (Zhang et al., 2026) spends three calls per epoch — Generator, Reflector, Curator — and merges the Curator's ADD/UPDATE/DELETE deltas deterministically, evicting past 40 bullets by track record rather than by age. *OpenEvolve* (OpenEvolve contributors, 2025) is the row for LLM-driven evolutionary search: a MAP-Elites archive whose cells hold one elite each, a parent drawn uniformly from one of three pools, and up to five inspirations shown alongside it. It is the only baseline here that keeps an archive of its own.

Coding agents. Claude Code and Cursor are measured as what they are — an agent alone with a shell, a checkout and an interpreter — rather than as a single prompt, since scoring them as a one-shot generator produces a number about the wrong object. Each gets its own sandbox built from the module's own testbed image, aligned to the same commit the search uses, with root and every tool enabled. Exactly two things are withheld: the repository's own test file for the module under test, since an agent that reads it is transcribing an answer, and web search and fetch, since the same tests are one query away. The deletion has to happen *after* the checkout is aligned, and `.git` has to go with it, or a single `git checkout` hands the answers back; both are asserted rather than assumed. Measurement happens in a different container from the one the agent works in, because the agent may edit the module or reinstall packages and none of that may reach what scores it. Suite versions are collected by watching the file rather than by replaying tool calls: with a shell available an agent writes with heredocs as readily as with an edit tool, and a transcript parser would silently miss those revisions — which, since the versions are the archive for these rows, would understate the method's own frontier.

G.2 PYNGUIN

Pynguin (Lukasczyk & Fraser, 2022) is the reference search-based generator for Python, and the row exists to compare against SBST measured rather than described.

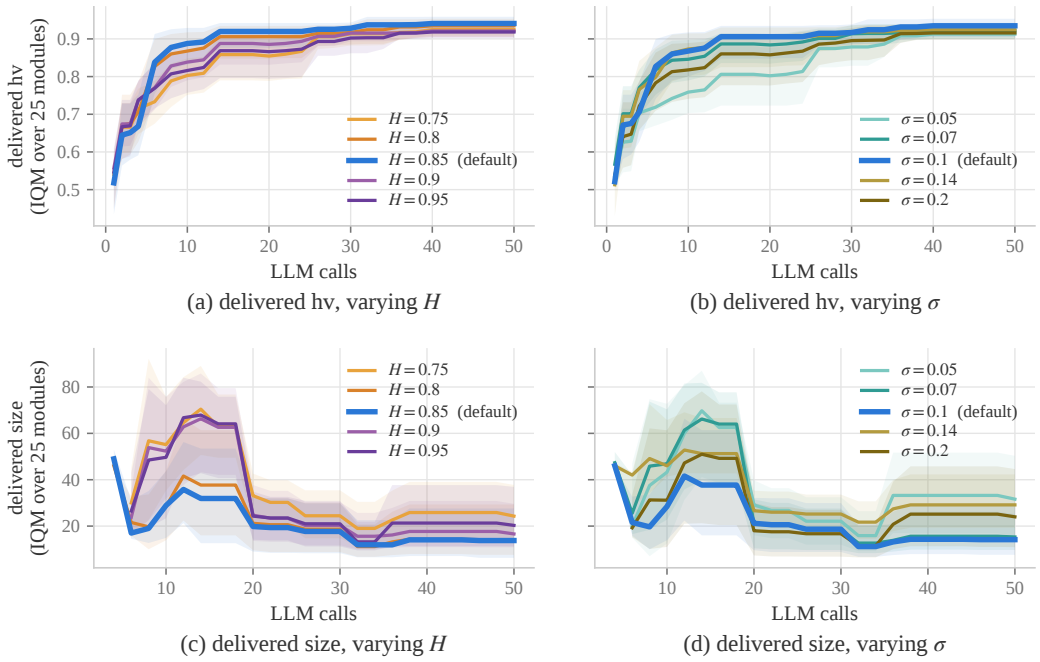
Eligible Modules. Pynguin 0.46.0 requires Python $\geq$ 3.10, while the TestGenEval testbeds were built with whatever interpreter each release needed — 3.6 through 3.11 across this benchmark, and 3.6 or 3.7 for most of the older django and sympy releases. Of the 129 modules, 42 run on Python 3.6 or 3.7, which no Pynguin release reaches, and are reported as not applicable rather than as a zero that would read like a bad score. Of the 87 that could be attempted, Pynguin returned no usable suite on 47: generation crashed, exhausted its wall clock, or could not load the module into its own process at all. Those failures are concentrated in exactly the code this benchmark is made of: metaclass-driven django model internals, sympy's dynamically dispatched printers, and modules whose import pulls in half a framework. They are therefore a property of SBST meeting real framework code rather than a harness artifact. The row is therefore a mean over the 40 modules Pynguin could be run and scored on, and the comparison is against Pynguin where Pynguin runs.

G.3 HYPERPARAMETERS

Table 7 lists the hyperparameters of TESEARCH. All are fixed before any reported run and shared by every module and every backbone. Three of them, the LMS step η, the EMA rate α and the confidence radius c_e, are common defaults and are not tuned. The remaining two, the spread σ of the transition kernel and the quality gate H of the acquisition, are calibrated rather than taken from a default, so for these two we run two experiments on a holdout: one selects their values (§G.3.1), and one tests how sensitive the results are to that choice (§G.3.2). The two do not earn their values the same way: H is chosen by that sweep, while σ is an estimate, the median per-run converged

Table 7: Hyperparameters of the search itself, with where each value comes from. The three ablations of Table 1 are reached by holding the ramp shut (*w/o* λ), by disabling the aim and crediting pulls to landings (*w/o kernel*), and by replacing the argmax with a round-robin sweep (*w/o UCB*); *w/o nearest-parents* changes the parent rule only.

Symbol	Value	Role	Origin
σ	0.10	landing spread, in log units; fixed, not learned	median of per-run converged fits on the holdout
η	0.2	LMS step for δ	common default; not tuned
α	0.3	EMA rate for $\widehat{Q}_j$	common default; not tuned
c_e	0.5	confidence radius on the tabular UCB width	common default (Bai et al., 2025); not tuned
H	0.85	quality gate; also sets the pressure ceiling $1 - H$	holds λ at 0 over the first ten calls (§5.4)



(a) delivered hv, varying H (b) delivered hv, varying σ

(c) delivered size, varying H (d) delivered size, varying σ

Figure 5: Robustness to the two calibrated constants, over the call budget, on the 25 holdout modules. **(a,c)** the five values of the quality gate H; **(b,d)** the five values of the landing spread σ. Curves are interquartile means over modules and bands are stratified bootstrap 95% CIs; the default setting is the same run in both columns and is drawn in the paper's blue.

residual spread (§4.1), so for σ the sweep confirms that the measured value is also not a bad one to optimize at.

G.3.1 Parameter selection

Experimental Setup. We set H and σ on a holdout of 25 modules, five from each of five repositories (django, sympy, scikit-learn, matplotlib, pytest), drawn from TestGenEval and disjoint from the evaluation set. σ is an estimate, the median per-run converged residual spread on this holdout; H is chosen by a sweep. Each constant takes five values with the other held at its default value, at the same budget $N = 50$: $H \in \{0.75, 0.80, \mathbf{0.85}, 0.90, 0.95\}$ and $\sigma \in \{0.05, 0.07, \mathbf{0.10}, 0.14, 0.20\}$, the latter evenly spaced on the log scale, so for σ the sweep confirms that the measured value is also a good one to run at. At the largest value, $H = 0.95$, the size-pressure ramp stays shut on most

modules. We report two delivered quantities over the call budget: hypervolume, and *delivered size*, the smallest archived suite that clears a quality level every setting reached on the module, which compares size at matched quality rather than penalizing a setting for delivering a stronger suite (§H). Curves are interquartile means over modules, the mean of the middle 50%, with 95% percentile bootstrap intervals (Agarwal et al., 2021). The resampling unit is the module and the resample is stratified by repository: each repository's modules are drawn with replacement to that repository's own count, so the five-way composition is held fixed, and the interquartile mean is recomputed over 2000 such resamples. Because the five repositories are fixed by design rather than sampled, the interval covers a fresh draw of modules within them, one run per module; it is not a statement about rerunning the same modules under different LLM sampling, for which one run per cell leaves no second level to resample. Budget points backed by fewer than three modules are dropped rather than plotted.

Results and Analysis. Neither constant is freely tuned on the evaluation set, which is what makes the sweep readable. H is at once the gate and the ceiling of Equation (9): pressure stays off until the running best hypervolume passes H, and once open it can never exceed $1 - H$, so a gate raised to protect quality for longer also weakens the anneal that must follow. The two ends of the sweep therefore fail for opposite reasons rather than sitting on one smooth knob. At $H = 0.95$ the median λ never leaves zero on the holdout, so that setting is the *w/o* λ ablation reached by another route and keeps whatever sizes the search happened to produce; at $H = 0.75$ the ramp opens on the fourth call, before the archive holds anything worth shrinking, and the acquisition is pulled toward niches that still contain nothing clearing the bar, which costs quality as well as size. The default setting opens the ramp on the ninth call and $H = 0.80$ and 0.90 on the eighth and the twenty-third, and that is the window the flat bottom of the curve corresponds to. For σ the result is an asymmetry. Understating the spread is the only setting the intervals punish: at $\sigma = 0.05$ the kernel gives 95% of a generation's credit to the niche it aimed at against 68% at the default value, so a suite landing one niche away updates $\widehat{Q}$ for a niche that was never exercised, and this is the one comparison in the figure whose bands separate. Overstating merely dilutes: $\sigma = 0.20$ spreads the same credit over three to four niches instead of two and costs almost nothing. The sweep therefore establishes neither value as optimal, and past the first half of the budget nothing separates at all; what it does establish is that the gate has a wide working window, that both failure modes are the designed ones rather than instabilities, and that the safe direction to misstate the actuator's spread is upward, which is the argument for estimating σ from the measured residual rather than tuning it. Two cautions on the mid-budget region: the humps there compare slightly different module sets, since a module enters the delivered-size average only once that setting has cleared the shared level on it, and the level itself is fixed from the completed runs, so delivered size is an analysis quantity rather than an anytime one.

G.3.2 Parameter sensitivity

Experimental Setup. The sweep above sets both constants on a holdout pooled over five repositories, which leaves open whether the values it selects are an artifact of that particular pool. We therefore read that same sweep once more for how far delivered quality moves across settings (Figure 5), and separately run the default configuration one repository at a time, with no constant varied and at the same budget $N = 50$. Figure 6(a–e) is one repository each, five modules apiece, and (f) pools all 25 and is therefore the same curve as the default setting of Figure 5. We report delivered hypervolume alone here: size is what the constants move, and the question this check asks is whether the quality the default setting delivers holds up repository by repository. The protocol is the one above with a single term removed. Curves are interquartile means with 95% percentile bootstrap intervals over 2000 resamples of modules, but a single-repository panel has no five-way composition left to hold fixed, so within it the resample is unstratified, and the interquartile mean over five modules is the mean of the middle three. Five is below the roughly ten runs down to which Agarwal et al. (2021) validate the procedure, so these intervals are wide by construction: the check can expose a sharp departure but not a subtle one, and we read it only for whether any repository contradicts the pooled curve.

Results and Analysis. Read for quality rather than size, the sweep is flat: no setting is separated from the others by more than its own sampling error at any point on the budget, so misplacing either constant costs delivered size and not delivered quality. Flatness on its own would carry little, since a panel too noisy to resolve anything looks the same, and the sweep settles that from inside. At $H = 0.95$ the ramp stays shut on most modules, so the *w/o* λ condition of Table 1 is already

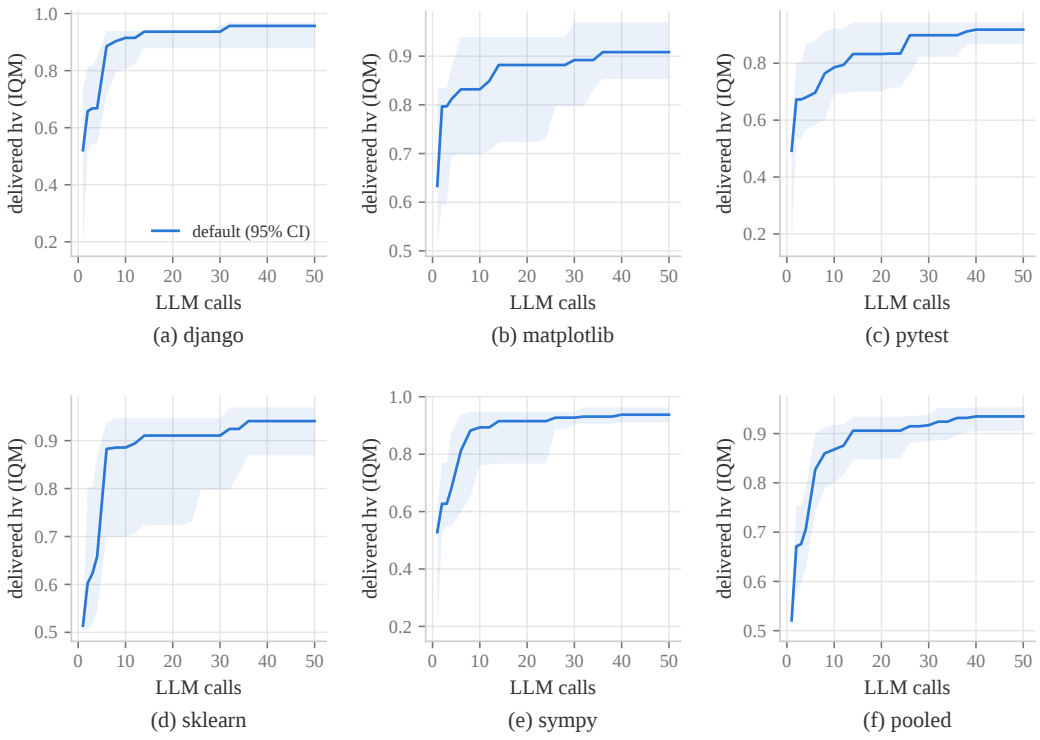


Figure 6: The default configuration one repository at a time, checking that its performance is consistent across repositories; no constant is varied. Line: the interquartile mean of the default setting; band: its bootstrap 95% CI over that repository's own five modules, on delivered hypervolume. **(a)–(e)** one repository each; **(f)** pools all 25.

present as one of the five settings, and it does sit at the bottom of the sweep on quality, by about as much as *w/o* λ gives up in Table 1, which is what §5.4 leads one to expect once λ is read as part of the acquisition rather than as a delivery-time filter. The setting that departs outright is $\sigma = 0.05$, early and on quality, and it is the one the kernel analysis says should fail, because it understates the actuator noise the search has to live with. A setting departing exactly where a mechanism predicts it is what licenses reading the rest as genuinely flat rather than merely unresolved. The per-repository panel (Figure 6) then carries the only claim in this appendix that reaches past the five repositories themselves, since the stratified bootstrap behind Figure 5 resamples modules inside repositories and holds the repositories fixed. It is a weak claim by construction, five modules to a panel and intervals wide to match: no repository contradicts the pooled curve, and those that depart most do so in the shape of the climb rather than in where it ends. That is consistent with the default configuration behaving the same way everywhere without being evidence that it does.

H Anytime behavior under a token budget

Figure 7 plots delivered hypervolume and delivered size against tokens actually spent, so every method is read at matched cost rather than at matched calls (refer to §F for the delivery rule). Among the single-artifact optimizers (Figure 7a), ACE plateaus earliest, within a percent of its final 0.692 by 226k tokens, because its Generator–Reflector–Curator loop refines one playbook whose operating point is fixed once the bullet list stops turning over; MIPROv2 reaches 0.746 by 298k as its categorical TPE exhausts the instruction $\times$ demonstration grid it was given; GEPA climbs longer, to 0.732 by 381k, since merging two Pareto-pool members keeps proposing instructions after single-candidate search would have stopped. Among the sampling and search methods, best-of-N saturates first, at 0.693 by 215k, which is the control it was included to provide: independent draws from one

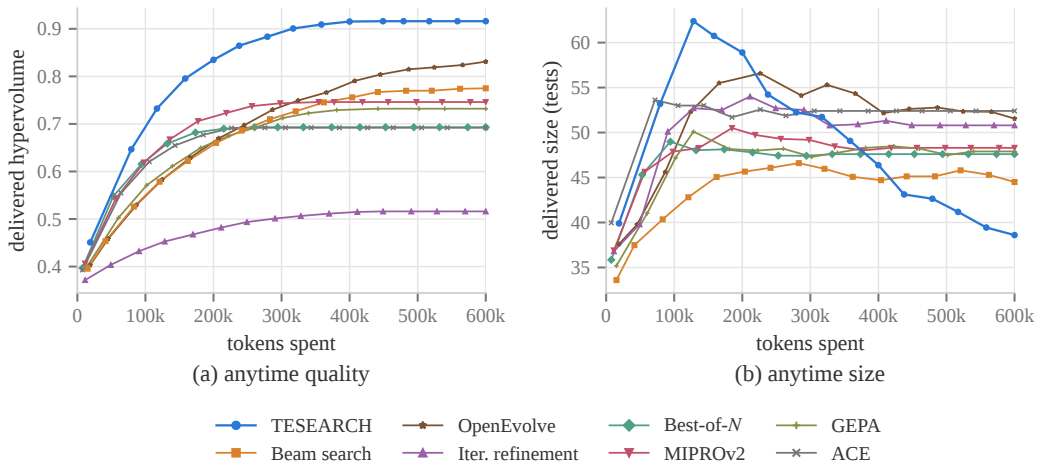


Figure 7: **Anytime behavior on the token axis.** Nothing is re-run: each curve is a replay of the runs behind Table 1, read at every point during a run for what would have been delivered had the budget stopped there, then averaged over those runs. The axis is tokens actually spent, so every method is read at matched cost rather than at matched calls. **(a)** Delivered hypervolume. **(b)** Delivered size, in tests written. Ours is the only curve that turns: it climbs while the search is still buying quality and falls once the size term anneals in, ending the smallest on the panel, while every baseline rises and then settles.

prompt stop paying once the sampling distribution is exhausted and no feedback crosses calls. Beam search improves until 483k and finishes at 0.775, the extra gain coming from a beam that a layer can never make worse. Iterative refinement is the weakest on the panel at 0.516 because a single chained trajectory keeps nothing to fall back on once the chain commits to a poor region. OpenEvolve is the nearest competitor at 0.831 and the only baseline still gaining at the end of the axis: it is the only baseline that keeps an archive, and an archive is what converts a long budget into sustained progress rather than early saturation. TESEARCH reaches 0.916, is ahead from the first reading and is never overtaken, matching what OpenEvolve delivers after its entire run at about a third of the budget and what the other six deliver between a fifteenth and a quarter of it. Panel (b) separates the two archives. Every baseline rises and then settles within about five tests of its own peak, and OpenEvolve ends among the largest suites on the panel despite its archive, because that archive is not indexed by suite size and so holds nothing smaller to hand back. Ours is the only curve that turns: size climbs to 62.4 tests at 128k while the quality gate holds the pressure term at zero, falls to about 43 by 449k as the ramp opens and quality levels off, and then falls a further four tests to 38.6 over the last quarter of the axis at an exactly unchanged hypervolume of 0.916. That final stretch is the mechanism in isolation, size bought at no cost in quality, and it is why we end with both the highest quality and the smallest suite on the panel.

I SUITE SIZE AS A PROXY FOR VERIFICATION COST

Suite size in this paper is the number of `test_` functions a suite contains (§F), and a smaller count is a cheaper verification signal only if the tests are not correspondingly longer or slower. A search rewarded for fewer tests could meet that reward by merging several tests into one, which lowers the count while leaving unchanged the code an agent must read and the time a gate must wait. The size floor $s_{\min}$ of §A rules out the extreme case, a whole module checked in a single test, but not a milder drift toward longer tests. Table 8 therefore reports three measures of cost for the delivered suites of Table 1: the test count, the lines of test code (**LOC**: non-blank, non-comment lines of the suite as written), and the wall-clock **time** of one run of the suite in a fresh container of its module, including container and interpreter start-up. LOC per test is the direct check on merging: had TESEARCH reached its smaller counts by packing checks into fewer tests, its tests would be longer than those of the baselines. We omit Pynguin: its suites are cheap only because they are weak (Table 1), whereas the question here is whether a suite can be cheap and strong at once.

Table 8: **Cost of the delivered suites** of Table 1, means over the 129 modules. **size**: number of tests, as in Table 1; **LOC**: non-blank, non-comment lines of test code; **LOC/test**: LOC divided by size; **time**: wall-clock seconds of one run of the suite in a fresh container, including container and interpreter start-up. Lower is cheaper in every column; bold = lowest per column. Claude Code runs on its native backbone only; — marks a cell that does not apply.

		GPT-5.4				Claude Opus 4.8				Gemini 3.1 Pro			
	Method	size	LOC	LOC/test	time	size	LOC	LOC/test	time	size	LOC	LOC/test	time
Inference-time scaling	Iter. refinement	50.8	816	16.1	5.71	72.5	705	9.7	6.83	56.8	765	13.5	5.94
	Best-of-N	47.6	645	13.6	5.08	61.5	494	8.0	5.62	51.4	577	11.2	5.21
	Beam search	44.5	668	15.0	5.34	68.8	601	8.7	6.27	52.6	661	12.6	5.33
	MIPROv2	48.3	774	16.0	5.46	63.1	614	9.7	6.05	53.1	716	13.5	5.40
	GEPA	47.9	768	16.0	5.89	67.2	646	9.6	6.48	54.7	728	13.3	5.77
	ACE	52.4	705	13.5	5.52	64.4	511	7.9	6.11	55.5	597	10.7	5.69
	OpenEvolve	51.5	868	16.9	5.97	68.0	665	9.8	6.71	56.6	779	13.8	5.86
Coding agents	Claude Code	—	—	—	—	51.6	429	8.3	4.37	—	—	—	—
	Cursor	44.4	**539**	**12.1**	4.62	48.2	353	**7.3**	4.96	37.0	**383**	**10.3**	4.18
	TESEARCH (full)	**38.6**	587	15.2	**3.74**	**29.1**	**254**	8.7	**3.91**	**36.3**	435	12.0	**3.66**

Results and Analysis. LOC and time track test count closely: across methods, their rank correlation with size lies between 0.80 and 0.98 on every backbone, so the smaller counts of Table 1 are smaller suites rather than recounted ones. Merging would show up as longer tests, yet LOC per test for TESEARCH is at or below the baseline median on every backbone, and below that of the strongest baseline on each (OpenEvolve, GEPA and beam search, respectively): against the method closest to it in quality, TESEARCH writes both fewer tests and shorter ones, so its smaller counts do not come from packing several tests into one. The one method with less test code is Cursor on GPT-5.4 and Gemini 3.1 Pro, and it buys that economy with shallower assertions: its tests are the shortest on the panel, and its mutation score trails TESEARCH's by a wide margin on both backbones (Table 1). No suite that comes close to TESEARCH's quality uses less code. Time gives the cleanest result of the three: TESEARCH executes fastest on every backbone, including the two on which Cursor writes less code, and time follows test count more closely than LOC does on all three. This is consistent with execution cost being driven by per-test overhead, such as collection and fixture set-up, more than by lines of code, which makes the number of tests the right quantity to minimize.

J CROSS-VERSION MIGRATION

This section details the migration experiment of §5.5.1: which instances qualify, how each migration instance is built, and what the agent receives and returns.

J.1 DATASET CONSTRUCTION

TestGenEval pins each module at a single release of its repository and defines no migration task, so we build one from the library's own history. The testbed image of every TestGenEval instance carries the repository's full git history, including release tags later than the pinned one. A migration instance is a forward port of one module within its own library: the module is taken at the pinned release, and the agent must rewrite it so that it works inside the next release, where every other file of the repository has already moved on, for example to renamed APIs, removed helpers or changed defaults. The target release is historical, so its code and tests exist; they are withheld from the agent and serve only to judge the result. The version that the maintainers wrote for that release is the reference answer, and it too is never shown to the agent.

An instance qualifies when its migration is well defined and can be scored without ambiguity; Table 9 lists the 12 that do. The module must have a next release: the target is the first release line after its pinned version, taking the newest patch tag of that line, and release lines are read from the repository's own git tags inside the testbed image, with prereleases excluded, rather than from a hard-coded version list, since tag spelling differs across repositories. The repository must also document the breaking changes at that boundary in its own release notes, since the agent receives them; sympy publishes none, so its modules do not qualify. The module's source must differ be-

tween the two releases, since a module that upstream leaves untouched has nothing to migrate and any condition would succeed trivially. The repository's test file for the module, which serves as the held-out tests, must exist at the target release, run in the target environment, contain at least 20 tests, and produce the same result on two runs of the upstream target module. These tests must also separate a migrated module from an unmigrated one. The upstream target module must pass every held-out test, so that a correct migration can succeed; and the source module, placed unchanged into the target release, must fail at least one of them, so that leaving the module as it is does not.

Table 9: The 12 migration instances. Each module is migrated from the release it is pinned to (*from*) to the newest patch of the next release line (*to*). Instance: the SWE-bench issue number of the TestGenEval instance. Tests: held-out tests at the target release. UDR (unmigrated detection rate): the fraction of held-out tests that fail when the source module is placed unchanged into the target release; a module that no longer imports there fails all of them.

Package	Instance	Module	From	To	Tests	UDR
django	11179	`django/db/models/deletion.py`	3.0.14	3.1.14	55	1.00
django	11422	`django/utils/autoreload.py`	3.0.14	3.1.14	75	0.15
django	11620	`django/views/debug.py`	3.0.14	3.1.14	83	0.46
django	11797	`django/db/models/lookups.py`	3.1.14	3.2.20	48	1.00
django	11848	`django/utils/http.py`	3.1.14	3.2.20	50	0.12
django	12125	`django/db/migrations/serializer.py`	3.1.14	3.2.20	49	0.12
django	12700	`django/views/debug.py`	3.1.14	3.2.20	91	0.23
django	14016	`django/db/models/query_utils.py`	4.0.10	4.1.10	28	0.57
django	14534	`django/forms/boundfield.py`	4.0.10	4.1.10	240	0.10
django	14580	`django/db/migrations/serializer.py`	4.0.10	4.1.10	51	1.00
django	14608	`django/forms/formsets.py`	4.0.10	4.1.10	157	0.11
sphinx	8627	`sphinx/util/typing.py`	3.5.4	4.0.3	21	1.00

Agent input and scoring. Besides the source module and the guide suite, the agent receives the evidence of what changed upstream at the release boundary: the breaking changes documented in the repository's release notes, at most 8 entries, and the names of the functions and classes the maintainers modified; the upstream replacement text is never shown. Every guide suite was generated against the source module and never saw the target release. For pass@5, after each attempt the agent receives the guide tests that fail on its module and their tracebacks, and the last module it returns is scored. A migration is scored by the fraction of held-out tests its module passes at the target release, and pass@1 and pass@5 average this rate over the 12 instances; a module that fails to import, or whose tests do not finish within 600 s, scores zero.

K Self-evolving agent verification

This section details the setup of the self-evolving agent experiment in §5.5.2: how the revision pool is built, how revisions are labeled and verified by humans, and how the gates are scored.

K.1 The revision pool

We revise eight modules of mini-swe-agent (SWE-agent team, 2025), chosen to cover the harness end to end: the agent loop, the local execution environment, the two action parsers, the cache-breakpoint policy, the message flattener, the model client, and the config merger (Table 2 gives the role of each). For each module, GPT-5.4 is shown the module source together with a revision intent and returns one complete revised module. Each module receives 32 revisions per round over two rounds, for 64 per module and 512 in total. Intents are drawn from the fixed list in Table 10, which covers the kinds of self-modification a harness-evolving agent proposes in practice. A behavior-preserving intent asks for the module to be improved without changing what it does; a behavior-changing intent asks for an edit that alters observable behavior. Table 11 shows representative revisions of each kind. Human verification (§K.2) removes 42 of the 512 candidates, leaving the 470 of Table 2. Of the 42, 11 are removed because the revised module fails to import, so neither gate can be run on it. This check is load-time only: a revision that imports and then raises when exercised is kept and labeled like any other, since raising where the original does not is exactly what a gate should catch.

Table 10: Revision intents. **Class** is what the intent asks for, not what the revision turns out to do; a revision enters Table 2 only if human verification confirms that it realizes its intent (§K.2). Intents come in pairs: each behavior-changing intent is the failure mode of the behavior-preserving intent above it, with the same goal and the same shape of edit. n counts revisions generated under each intent, pooled over the eight modules.

Family	Intent	Class	n
Performance	cache a function whose result depends only on its arguments	preserving	16
	cache a function that reads changing state (clock, counter, file)	changing	16
	list to set, where order and duplicates are unused	preserving	16
	list to set, where order or duplicates are relied on	changing	16
	batch independent calls, keeping results and their order	preserving	16
	batch calls, changing which error surfaces and when	changing	16
	return early where no side effect follows	preserving	16
	return early, skipping a side effect (log, counter, cleanup)	changing	16
Token	shorten the output, keeping every field a reader uses	preserving	16
	shorten the output, changing a key or separator a reader parses	changing	16
	truncate long output, keeping head and tail with a marker	preserving	16
	truncate long output, keeping only the head	changing	16
	stop emitting a field that no code reads	preserving	16
	stop emitting a field read only on a rare path (e.g., errors)	changing	16
	merge repeated outputs that are byte-identical	preserving	16
	merge outputs that print alike but differ (e.g., exit code)	changing	16
Robustness	retry a call that is safe to repeat	preserving	16
	retry a call whose repetition has an extra effect	changing	16
	add a timeout that reports expiry as a failure	preserving	16
	add a timeout that returns partial output as complete	changing	16
	narrow `except Exception` to every type the block raises	preserving	16
	narrow `except Exception`, missing a type the block raises	changing	16
	handle `None` where the value is legitimately optional	preserving	16
	return `None` silently where the code should raise	changing	16
Structure	extract a helper, keeping captured variables and defaults	preserving	16
	extract a helper, changing a captured variable or default	changing	16
	group parameters into a fresh object per call	preserving	16
	group parameters into an object shared across calls	changing	16
	merge two branches with identical bodies	preserving	16
	merge two nearly identical branches, losing the difference	changing	16
	add type hints and docstrings only	preserving	16
	add type hints plus a small change to a conversion or default	changing	16
Total			512

K.2 Label construction and human verification

Labels are built in three steps. First, the intent fixes what a revision is meant to do: a behavior-preserving intent asks for an edit that leaves every observable behavior of the module unchanged, and a behavior-changing intent asks for one that alters it (Table 10). Second, GPT-5.4 implements the intent as a complete revised module (§K.1). An intent is a request rather than a guarantee: the model can break behavior while trying to preserve it, or produce a no-op while trying to change it. Third, human annotators check every revision against its intent and remove those that do not realize it. Each remaining revision is labeled divergent if its intent is behavior-changing and equivalent otherwise. The label is fixed before either gate is run, and neither gate contributes to it.

Two annotators, each with a computer-science background and at least three years of Python experience, independently inspected all 512 revisions. For each revision, an annotator first imported the revised module to check that it loads. The annotator was then shown the intent the revision was generated under, the original module, the revised module and their unified diff, and judged whether the revision realizes its intent. For a behavior-preserving intent, this means that the revision leaves

Table 11: Example divergent and equivalent revisions for six of the eight harness modules, with the revision intent and content. *eq* and *div* abbreviate equivalent and divergent.

Module	Intent	What the revision did	Label
`agents/default`	narrow `except Exception`, missing a type the block raises	narrows `except Exception` around the model call to `ValueError` and `TypeError`; a model raising `KeyError` or `OSError` now escapes the loop	div
`environments/local`	return early, skipping a side effect (log, counter, cleanup)	checks for the submission sentinel only when the command failed, so a successful submit no longer ends the episode	div
	extract a helper, keeping captured variables and defaults	moves the decoding of a failed command's partial output into a module-level helper, called at the same point on the same exception	eq
`actions_toolcall`	list to set, where order and duplicates are unused	tests for the required `command` argument with a one-element set's `issubset` instead of `in`	eq
`actions_text`	cache a function that reads changing state (clock, counter, file)	reads the clock once at import, so every observation carries the same stale timestamp	div
`cache_control`	stop emitting a field read only on a rare path (e.g., errors)	deletes both places that attach the ephemeral `cache_control` marker, so no breakpoint is ever set	div
	cache a function whose result depends only on its arguments	caches the content-text lookup behind a function that returns its argument, so a cache hit returns what a fresh call would	eq
`serialize`	merge two branches with identical bodies	folds the merge-into-existing and fresh-copy branches into one call, passing `None` for a missing previous value, which the merge already skips	eq

what the module returns, raises, writes and calls unchanged on every input the harness can pass it; for a behavior-changing intent, that it changes at least one of these on some such input. A third annotator resolved the revisions on which the two disagreed.

The two annotators agree on 95.51% of revisions before reconciliation (Cohen's $\kappa = 0.74$), and the third annotator resolves the remaining 23. Of the 512 revisions, 42 (8.2%) are removed: 11 because the revised module fails to import, and 31 because the implementation does not realize its intent. The remaining 470 form Table 2. Behavior-preserving revisions that miss their intent are mostly edits that change a value the intent assumed unused, such as a cached clock read or a field read on a rare path. Behavior-changing revisions that miss theirs are mostly edits whose change never takes effect, such as an early return that skips no work.

K.3 Metrics

Each gate makes one accept-or-reject decision per revision. A gate's suite is first run on the unmodified module to record the tests that already fail there; the gate then rejects a revision if running the suite on it produces a failure outside that set. A gate that cannot run on the revision at all, or does not finish within 120 s, also rejects it, since the agent receives no answer it could act on. Let $\mathcal{D}$ and $\mathcal{E}$ be the divergent and equivalent revisions of a pool, and $\mathcal{R}$ the revisions a gate rejects.

Divergent-revision detection rate is the fraction of divergent revisions the gate rejects,

$$\mathrm{DDR} = \frac{|\mathcal{R} \cap \mathcal{D}|}{|\mathcal{D}|}.$$

A divergent revision the gate accepts changes the harness's behavior without the agent noticing, and every later self-modification builds on it. Higher is better.

Equivalent-revision false-alarm rate is the fraction of equivalent revisions the gate rejects,

$$\mathrm{FAR} = \frac{|\mathcal{R} \cap \mathcal{E}|}{|\mathcal{E}|}.$$

An equivalent revision the gate rejects is a sound improvement the agent is forced to discard, and a gate with a high false-alarm rate stalls evolution. Lower is better.

Verification cost is the wall-clock time of one decision: running the gate's suite on one revision inside a fresh container, including container start-up, which both gates pay alike. Table 2 reports the mean over a module's revisions and, for the pooled row, over all decisions. Neither gate calls a model at decision time, so a decision costs no tokens.