SPEND COMPUTE WHERE ERRORS PROPAGATE: RELIABILITY BUDGETS FOR DEPENDENT TASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

An error in a shared intermediate result can invalidate many downstream outputs. We derive a rule for allocating inference compute according to these unequal consequences. For dependent sub-problems with partial credit, downstream reach weights a separable upper bound on expected invalidated work. Under independent node errors and exponential error curves, optimizing this bound gives a water-filling allocation with a finite guarantee relative to the true-loss optimum. For homogeneous rates and unclipped allocations, its surrogate gain over uniform is exactly the arithmetic-to-geometric-mean ratio of the reach weights. On synthetic trees, median excess loss over the exact optimum is 1.3%–7.3%; a separate timing study finds 34–$1{,}300\times$ faster allocation than exact optimization. On selected, moderate-difficulty arithmetic trees at tight budgets, reach-weighted allocation outperforms an equally adaptive structure-blind policy in all three models (paired gain 1.08–$1.13\times$). On code, idealized reference-code repairs show that reach times a local failure signal obtains 71–88% of the best single repair's benefit, versus 43–68% for the signal alone, among states where a repair helps. Actual code savings largely come from adaptive stopping; reach adds setting-dependent gains and degree performs similarly. The framework connects an efficient allocation rule to measurable gains and explicit conditions for when dependency information is useful.

1 INTRODUCTION

A code generator has enough budget for five more attempts, but several functions still fail tests. One is a shared parser called by many other functions; another formats a single output. Fixing either counts as one local success, yet fixing the parser may restore several outputs at once. A policy that considers only which function is most likely to be wrong can miss this difference. Where should the next attempt go?

This choice arises whenever a task is solved through dependent intermediate steps. A multiplication result may be reused in several later calculations, just as a helper function may serve several callers. Repeated sampling can improve a step's reliability by selecting among candidates with voting (Wang et al., 2023) or tests. Under a fixed budget, however, another sample at one step means one fewer elsewhere. Equal effort per step is easy to implement, but it ignores how much work depends on each answer. Even steps with the same error rate can deserve different budgets.

The objective matters too. If useful parts retain value when others fail, an error should be assessed by how much work it invalidates. We study this setting with a given dependency graph: a wrong intermediate answer invalidates every node that depends on it (Figure 1). This turns the intuitive importance of a shared result into a quantitative weight, its downstream reach. More reach can justify more samples, but the allocation must also account for difficulty and diminishing returns. We derive a rule that balances these factors, bound its loss relative to the true optimum, and characterize when dependency weights improve on uniform allocation. We also show why, under our error model, the graph can cease to matter when every step must be correct.

Using this rule requires distinguishing two decisions: which component is valuable to correct, and how to correct it. We call these *targeting* and *repair*. Dependency information can help with the first even when another sample rarely achieves the second. Our experiments test both: controlled arithmetic comparisons isolate the contribution of dependency weights, while code experiments measure

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.

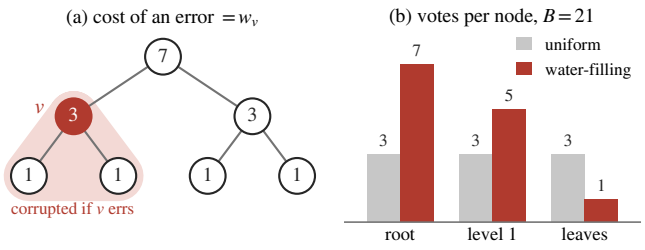

Figure 1: **(a)** In the propagation model, an error at a node invalidates its subtree, whose size is w_v. **(b)** With 21 votes on this seven-node tree and a common error-decay rate $c = 0.4$, water-filling (Theorem 3) gives $6.4, 4.3, 4.3, 1.5, 1.5, 1.5, 1.5$ votes. The continuous surrogate gain over uniform is $\mathrm{AM}(w)/\mathrm{GM}(w) = 1.344$; the displayed odd rounding gives 1.307.

repair value and the improvements that resampling actually delivers. This distinction connects the allocation theory to systems with observed votes, imperfect tests, and finite candidate pools.

Contributions. We connect propagated inference errors to an efficient allocation rule and a finite guarantee for true loss.

1. **An allocation rule with a finite guarantee.** Reach weights a separable upper bound on expected invalidated work. Under independent node errors, optimizing it approximates the true-loss optimum. Exponential error curves give water-filling and, for homogeneous rates without clipping, an exact AM/GM gain law. Exact synthetic comparisons measure approximation quality and runtime.
2. **Controlled evidence for dependency weights.** On selected arithmetic trees at tight budgets, reach-weighted allocation outperforms an equally adaptive blind policy in all three models, with paired intervals excluding one. Shuffling weights reduces performance. Static graph-based allocation achieves similar gains to the tested adaptive and calibration-oracle policies without node-specific calibration or online feedback.
3. **Repair priorities and simpler alternatives.** Idealized reference-code repairs show that reach improves target selection over a local failure signal on author-built tasks and ClassEval. Actual code savings largely come from adaptive stopping. Degree performs similarly to reach; a proxy-weight corollary quantifies the cost of simpler weights under the surrogate.

Roadmap. Section 3 defines the model; Sections 4–6 develop the theory. Section 7 presents the experiments and Section 8 their implications. Proofs and full protocols are in the appendix.

2 Related Work

Self-consistency and adaptive stopping. Wang et al. (2023) introduced majority voting over sampled reasoning chains, and early-stopping variants add statistical guarantees (Cordero-Encinar & Duncan, 2025; Aghazadeh et al., 2025; Huang et al., 2026). They operate on a single query. Talebirad et al. (2026) study crowds of several models on one query. We allocate a budget across the dependent sub-problems of one task.

Test-time compute allocation. Snell et al. (2025) show that adaptive test-time compute can outperform increasing model size, and Chen et al. (2024) derive scaling laws for flat voting. DORA (Wang et al., 2025b) allocates rollouts optimally across reasoning directions for final-answer success. Tree-of-Thoughts (Yao et al., 2023) and Graph-of-Thoughts (Besta et al., 2024) choose which branch to expand, and recent systems allocate compute across queries or pipeline stages by heuristics (Cui et al., 2025; Lin et al., 2025; Li et al., 2026; Wan et al., 2026; Wang et al., 2025a). Our setting is a fixed decomposition with propagated errors and partial credit for correct work. The resulting objective weights an error by its downstream consequences, rather than treating reasoning directions as alternative routes to a single correct answer.

Sequential decomposition. MAKER (Meyerson et al., 2025) solves million-step chains with all-or-nothing loss and uniform per-step voting, which motivates our all-or-nothing comparison. Theorem 5 gives uniform allocation for homogeneous independent exponential error curves; these assumptions need not describe that system exactly.

Redundancy allocation and sequential sampling. Unequal error protection (Borade et al., 2009), optimal computing budget allocation (OCBA) (Chen et al., 2000), and reliability redundancy allocation (Barlow & Proschan, 1965; Kuo & Prasad, 2000) study how to distribute resources across units. Weighted exponential objectives yield the classical water-filling calculation used here. Fault trees (Vesely et al., 1981) and cascading failures (Motter & Lai, 2002) model propagation, and errors also propagate through hierarchical classifiers (Silla & Freitas, 2011). Our contribution connects these classical allocation tools to propagated inference errors: reach determines the weights, path error mass bounds approximation of true loss, and the objective determines whether graph structure matters. Matched adaptive comparisons test the value of the resulting weights.

3 SETUP

Decomposition graph. A task is decomposed into n sub-problems, the nodes of a rooted tree $\mathcal{T} = (V, E)$ that is given rather than learned (DAGs are treated in §4.1). The subtree of v is v with all its descendants, and $w_v = |\mathrm{subtree}(v)|$. An error at v *corrupts* every node of $\mathrm{subtree}(v)$.

Selectors. Node v receives $b_v \geq 1$ samples and a selector returns one answer; the budget is $B = \sum_v b_v$. We write $\varepsilon_v(b)$ for the probability that the selected answer is wrong when the node's input is correct, and consider three selectors. *Verifier selection* keeps a sample that passes a test. If samples are i.i.d. and pass with probability $q_v \in (0, 1)$, no sample passes with probability exactly $(1 - q_v)^b$, so $\varepsilon_v(b) = e^{-c_v b}$ with $c_v = -\ln(1 - q_v)$, where the error is measured against the test; this identity describes test failure, not semantic correctness. False acceptance can leave residual errors (§4.1). *Binary majority* returns the majority of b i.i.d. correct/incorrect votes with accuracy q_v; for $q_v > \frac{1}{2}$ its error is at most $e^{-c_v b}$ with $c_v = 2(q_v - \frac{1}{2})^2$, and for $q_v \leq \frac{1}{2}$ more votes do not help. *Plurality* returns the most frequent of b i.i.d. answers; its error decays exponentially under a positive gap between the correct answer and each wrong one, though not as a single exponential at small b.

Remark 1 (Parity of binary majority). For binary majority over i.i.d. votes with ties broken by a fair coin, $\varepsilon_v(2k) = \varepsilon_v(2k-1)$ for all $k \geq 1$ and all q_v (proof in Appendix A.1). A comparison at an even per-node budget therefore rewards any policy that happens to create odd counts. The identity does not hold for plurality over many answers, for deterministic tie-breaking, or for verifier selection.

Assumption 1 (Exponential error curve)**.** $\varepsilon_v(b) = A_v e^{-c_v b}$ with $c_v > 0$ and $A_v \in (0, 1]$, optionally plus a floor $\varepsilon_{\infty,v}$ (§4.1), with parameters chosen so $0 \leq \varepsilon_v(b) \leq 1$.

For verifier selection with i.i.d. samples the assumption holds with $A_v = 1$ relative to the test. For voting it is an approximation that we fit where needed; our experiments never rely on it for scoring.

Losses. The *corrupted-subtree loss* is $\mathcal{L}(b) = \mathbb{E}[\#\text{corrupted nodes}]$, where u is corrupted if u or an ancestor of u errs. Its separable *surrogate* is $L(b) = \sum_v w_v \varepsilon_v(b_v)$. The corrupted-subtree loss grants partial credit and fits outputs such as collections of intermediate answers or library components whose correct parts retain value. The *all-or-nothing* loss $\mathcal{L}_{\mathrm{aon}}(b) = P(\text{some node errs})$ fits tasks that succeed only when every component is correct. In both, ε_v describes v on a correct input; every descendant of an incorrect node counts as corrupted, regardless of its own answer.

Information available to policies. We specify what each policy may observe (Table 1). *Deployable* policies see the graph, the answers or test outcomes of the samples they draw, and a prior fixed before the experiment, either from disjoint calibration problems or chosen a priori. *Oracle* policies also see per-node difficulty estimated from samples not charged to their budget. We report them only as references.

Table 1: Policies and the information each may use. "Own draws" means the answers (voting) or selection-test outcomes (code) of samples the policy has drawn, all charged to B.

Policy	Allocation rule	Information beyond the graph
uniform	B/n per node	none
weight-only	$b_v \propto w_v$ (one sample minimum)	none
structure-only	water-filling at a fixed nominal rate	fixed prior
adaptive warm start	m votes per node, then adaptive	own draws, fixed prior
adaptive, blind	largest estimated error reduction	own draws, fixed prior
adaptive, reach	same, weighted by reach w_v	own draws, fixed prior
adaptive, degree/shuffled	direct degree or permuted reach	own draws, fixed prior
difficulty-only (oracle)	$b_v \propto 1/\hat{c}_v$	per-node curves from uncharged samples
water-filling (oracle)	Theorem 3 on $(\hat{c}_v, w_v \hat{A}_v)$	per-node curves from uncharged samples
exact static optimum	exact tree dynamic program	evaluation error curves

4 Allocation under a known error model

The surrogate sums the number of nodes each error could corrupt. It can therefore count a corrupted descendant more than once. The next result bounds this overcounting using the error mass along a dependency path.

Lemma 1 (Path-limited sandwich). *Let $S_v = \sum_{u \preceq v} \varepsilon_u(b_u)$ be the error mass on the path from the root to v, let $\gamma_{\mathrm{path}}(b) = \max_v S_v$, and let $\psi(g) = (1 - e^{-g})/g$ with $\psi(0) = 1$. Then $\mathcal{L}(b) \le L(b)$ always, and if node errors are independent, $\psi(\gamma_{\mathrm{path}})\, L(b) \le \mathcal{L}(b)$. Moreover ψ is decreasing and $\psi(g) \ge 1/(1+g)$.*

Corollary 2 (Surrogate optimality). *If $b^\star$ minimizes L and b^{true} minimizes $\mathcal{L}$ under independent node errors, then $\mathcal{L}(b^\star) \le \phi\big(\gamma_{\mathrm{path}}(b^{\mathrm{true}})\big)\mathcal{L}(b^{\mathrm{true}})$ with $\phi = 1/\psi \le 1 + \gamma_{\mathrm{path}}$.*

This compares the *true losses* of the surrogate-optimal and true-optimal allocations over the same feasible set. The factor is finite at every error level and equals $1 + \gamma_{\mathrm{path}}/2 + O(\gamma_{\mathrm{path}}^2)$ near zero. On a tree it depends on depth times per-node error, rather than n times per-node error. Using the largest path error mass makes the bound conservative; its dependence on the unknown true optimum prevents direct evaluation during deployment. Independence is needed for the lower bound and approximation guarantee. The upper bound $\mathcal{L} \le L$ remains valid for dependent node errors by the union bound.

We optimize a continuous relaxation with $b_v \ge 0$ and $A_v = 1$; general amplitudes replace w_v by $w_v A_v$. Every node with positive allocation then has the same marginal return, giving the classical water-filling form. Experiments enforce at least one sample per node and integer budgets; a lower bound ℓ_v replaces the zero in the formula by ℓ_v.

Theorem 3 (Optimal allocation). *The unique minimizer of $L(b) = \sum_v w_v e^{-c_v b_v}$ subject to $\sum_v b_v = B > 0$, $b \ge 0$ is $b_v^\star = \max\big(0,\, c_v^{-1} \ln(c_v w_v/\lambda)\big)$, with $\lambda > 0$ fixed by the budget.*

Corollary 4 (AM/GM law). *If $c_v \equiv c$ and no node clips to zero, then $b_v^\star = B/n + c^{-1} \ln(w_v/\mathrm{GM}(w))$ and*

$$\frac{L(\mathrm{uniform})}{L(b^\star)} = \frac{\mathrm{AM}(w)}{\mathrm{GM}(w)} \ge 1,$$

with equality iff all w_v are equal, which for a rooted tree forces $n = 1$.

The ratio depends only on the graph. It grows as $\Theta(\log n)$ on balanced trees, is below 2 on stars, and tends to $e/2 \approx 1.36$ from below on chains. It describes the surrogate at continuous, unclipped allocations. Integer rounding, clipping, exact vote-error curves, and error compounding along paths all change the realized gain, and §7.1 measures each effect separately. With heterogeneous rates the allocation also allocates more samples to difficult nodes, and the ratio is no longer an upper bound on the gain.

4.1 FLOORS, COSTS, AND DAGS

Floors. Systematic model errors and imperfect tests can leave residual error as the budget grows. We model a budget-independent residual explicitly as a floor: $\varepsilon_v(b) = \varepsilon_{\infty,v} + A_v e^{-c_v b}$.

The floor contributes a constant to the surrogate, so the allocation still water-fills the reducible coefficients $w_v A_v$. As the floor dominates, the achievable gain approaches one (Proposition 8, Appendix A). For the exact path-product loss, floors can also change the optimum.

Token costs and DAGs. With per-sample costs κ_v the optimum water-fills $w_v A_v/\kappa_v$ (Proposition 9, Appendix A). On a DAG in which an error at v corrupts every node that depends on v, the same results hold with $w_v = |\mathrm{reach}(v)|$, the number of such nodes including v, and with γ_{path} replaced by the largest error mass over a node's ancestor set (Proposition 10). On a DAG that set can be much larger than any single path.

5 WHEN THE GRAPH DOES NOT MATTER

Theorem 5 (Allocation under two loss functions)**.** *Fix a tree, a continuous budget $\sum_v b_v = B > 0$, $b_v \geq 0$, and $\varepsilon_v(b) = e^{-c_v b}$ with independent node errors. (1) Under the corrupted-subtree surrogate the optimum is the reach-weighted water-filling of Theorem 3. (2) Under all-or-nothing loss $1 - \prod_v (1 - e^{-c_v b_v})$ the unique optimum is $b_v = c_v^{-1} \log(1 + c_v/\lambda)$, which does not depend on the graph. If the rates are also homogeneous it is uniform, and every non-uniform allocation is strictly worse.*

Under this model, all-or-nothing loss rewards avoiding every local error, regardless of how many descendants an error would corrupt. The result applies to independent node errors with fixed error curves; uniformity also requires homogeneous difficulty. It neither establishes optimality for adaptive code generation nor implies that stopping alone suffices. Uniform per-step voting in MAKER (Meyerson et al., 2025) is consistent with this boundary case, without establishing that its error process satisfies these assumptions.

6 UNKNOWN DIFFICULTIES

Water-filling needs each node's rate, which is not known before sampling.

Proposition 6 (Break-even probe count)**.** *Let rates be homogeneous, $c_v \equiv c$, and suppose water-filling at both budgets B and $B - mn$ is unclipped. Spending m samples per node on probes that are then discarded and water-filling the rest with exact rates gives surrogate loss $L^\star(B)e^{cm}$, which is below the uniform allocation's $L_{\mathrm{unif}}(B)$ iff $m < c^{-1} \ln(\mathrm{AM}(w)/\mathrm{GM}(w))$.*

For $\mathrm{AM}/\mathrm{GM} = 1.5$ and $c = 0.3$, the threshold is 1.35 probes per node: only one discarded probe per node can leave an advantage over uniform. Reusing probes keeps their votes available, but estimating the error rates still has a cost. Appendix A.10 bounds the surrogate error of a plug-in allocation using labeled probes. Its sufficient sample size exceeds a hundred probes per node, and it does not cover label-free voting or the small budgets studied here.

We therefore evaluate simple policies that reuse every draw. For arithmetic, the adaptive policy starts with one vote per node and assigns each next pair to the largest estimated reduction $w_v[\hat{\varepsilon}_v(b_v) - \hat{\varepsilon}_v(b_v + 2)]$. Agreement among observed answers and a fixed prior supply the estimate. For code, it samples an unresolved component with the largest $w_v/(a_v + 2)$, where a_v counts additional attempts; this is a diminishing-return heuristic, not a calibrated error probability. A *blind* counterpart sets $w_v \equiv 1$, a *degree* counterpart counts direct dependents, and a *shuffled* control permutes reach weights. These matched comparisons isolate the contribution of structure (Algorithm 1, Appendix B).

7 EXPERIMENTS

We test allocation quality against an exact optimum, dependency weights on arithmetic, and repair priorities on code, then examine executed propagation and boundary cases. Deployable policies

Table 2: Mean gain of water-filling over uniform on the same set of unclipped synthetic configurations as each idealization is removed.

	AM/GM	Continuous surrogate	+ Odd rounding	+ Exact binomial	+ True loss
$B = 3n$ (10 configurations)	1.76	1.76	1.75	1.60	1.45
$B = 5n$ (18 configurations)	1.86	1.86	1.85	1.63	1.51

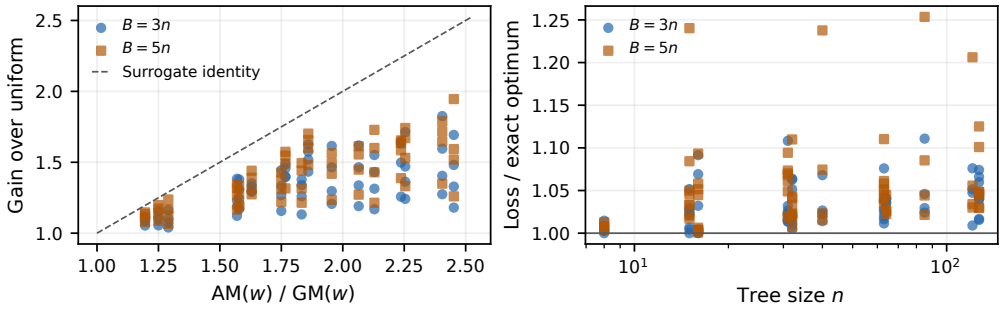

Figure 2: Synthetic trees, exact binomial errors, odd rounding. Left: realized gain against $\mathrm{AM}(w)/\mathrm{GM}(w)$. Right: water-filling's true loss divided by the certified optimum of Proposition 13.

pay for every sample; oracle and idealized results are labeled separately. All static allocations use odd vote counts to control the parity effect (Remark 1); sequential policies may stop at any count. Appendix B records splits, priors, eligibility rules, and protocol deviations. Analyses conducted after the main results are labeled exploratory.

7.1 How close is water-filling to the optimum?

We use 17 tree shapes with 8 to 127 nodes, homogeneous accuracy q, and exact binomial vote errors. Appendix B.1 also quantifies the spurious gains from even-vote uniform baselines.

Predicted and measured gains. Table 2 follows the same unclipped configurations (10 of 68 at $B = 3n$ and 18 of 68 at $B = 5n$) as each idealization is removed. The continuous surrogate ratio equals $\mathrm{AM}(w)/\mathrm{GM}(w)$ to 2×10^{-15}; odd rounding changes it little. Exact binomial tails and overlapping corruption along paths account for most of the reduction. Including clipped allocations, the mean realized gain over all 68 configurations is 1.34 at $B = 3n$ and 1.40 at $B = 5n$. It exceeds one in every configuration and correlates with AM/GM at $r = 0.69$.

Distance to the optimum. The exact dynamic program (Proposition 13) minimizes true loss over odd allocations. Water-filling's excess loss is 1.3%–7.3% at the median and 25.4% in the worst case (Figure 2, right). The optimum gains 1.22–1.64× over uniform, against water-filling's 1.20–1.50×. Median path error mass at the optimum is 0.35–1.36. These exact-binomial comparisons measure allocation quality beyond the continuous exponential model. The exact program requires a tree, full per-node error tables, and $O(nB^2)$ time. Water-filling also applies to DAGs and is 34–1,300× faster in our 15–511-node timing study (Appendix B.1).

7.2 Arithmetic: the graph as a static prior

Each node is a multi-digit multiplication problem, scored by plurality over raw integer answers. For each model, 192 new problems receive 8 calibration answers each. Sixteen randomly selected problems supply the policies' label-free prior and never enter evaluation trees. Evaluation problems have 3–7 correct calibration answers and are votable on them, leaving 31/17/17 problems for Llama-3.1-8B, Qwen2.5-7B, and Qwen2.5-14B. Each receives 16 new answers used exclusively for scoring and online draws. We adopted this calibration-only difficulty filter after an initial pool was dominated by problems solved on every sample or almost never; on its unfiltered problems, every policy

Table 3: Arithmetic trees, gain in expected corrupted nodes over uniform at matched total samples ($B = 3n$). "Reach/blind" is the paired ratio of the blind adaptive policy's loss to the reach-weighted one's (95% bootstrap CI). The last column is the gain of the exact best static allocation for the estimated evaluation curves. It is not an upper bound on adaptive policies, and no policy can see it.

	deployable					reference
Model	structure-only	adapt. blind	adapt. shuffled	adapt. reach	reach/blind	static opt.
Llama-3.1-8B	1.13	0.99	0.95	1.12	1.13 [1.09,1.16]	1.26
Qwen2.5-7B	1.07	1.03	0.97	1.11	1.08 [1.05,1.11]	1.17
Qwen2.5-14B	1.12	1.02	0.96	1.14	1.12 [1.08,1.16]	1.23

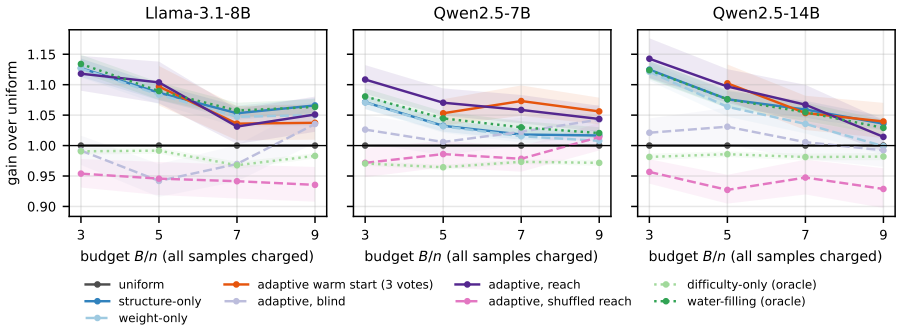

Figure 3: Arithmetic trees: gain over uniform as the budget grows, with every sample charged. Shaded bands are 95% bootstrap intervals over assignments. Solid lines are deployable, dotted lines are oracle references.

is within a few percent of uniform (Appendix C.1). Problems are assigned randomly, independently of position, to thirteen topologies with 5–63 nodes.

Table 3 and Figure 3 summarize the results. Under uniform voting, the fraction of corrupted nodes is 0.55–0.60. The exact best static allocation for the estimated evaluation curves reduces this loss by a factor of 1.17–1.26×. This provides a reference for the gains available to static allocation.

Effect of dependency weights. Reach-weighted adaptivity outperforms the equally adaptive blind policy in all three models: the paired gain is 1.08–1.13×, with every 95% interval excluding one. The reach-weighted policy gains 1.11–1.14× over uniform, compared with 0.99–1.03× for blind adaptivity. Shuffling the reach weights within each tree gives 0.95–0.97×, and true reach improves on shuffled weights by 1.14–1.19×. Thus the gain depends on assigning weights to the correct nodes. Direct out-degree also improves allocation, with reach ahead by 1.02–1.04×.

Static versus adaptive allocation. Structure-only water-filling, which uses no per-node information and no feedback, gains 1.07–1.13×, and the paired ratio of its loss to the reach-weighted adaptive policy's is 0.99–1.03×. These gains require no node-specific calibration or online feedback; the nominal rate uses a prior from disjoint calibration problems. The advantage shrinks as the budget grows and is 1.00–1.02× at $B = 9n$. Oracle water-filling uses 8 uncharged calibration answers per problem and gains 1.08–1.13×, similar to structure-only allocation. For this oracle, gains predicted from calibration-fitted curves correlate with gains measured on held-out answers across the thirteen topologies ($r = 0.80$–0.91). This tests prediction across answer samples, not generalization to new problems.

These intervals describe resampling over assignments of 31, 17, and 17 selected problems; they do not cover uncertainty about new problems.

7.3 Code: repair priorities and resampling

Unit tests provide feedback and call graphs give the weights. Under our *realistic verifier*, development tests use each policy's current generated dependencies; separate final tests score saved assem-

Table 4: Whole-task success at the $3\times$ token checkpoint of $4\times$-cap runs (task counts in parentheses), and the paired reach−blind difference in percentage points with a task-level bootstrap interval. Token ratios, component fractions, all policies, and sensitivity analyses are in Appendix C.6.

	whole-task success				
Setting	uniform	adapt. blind	adapt. degree	adapt. reach	reach−blind (pp)
Pilot suite, Qwen (16)	0.51	0.56	0.56	0.56	$+0.0$ $[+0.0, +0.0]$
Pilot suite, Llama (16)	0.32	0.44	0.47	0.47	$+3.1$ $[-0.6, +7.5]$
Pilot ClassEval, Qwen (53)	0.28	0.36	0.35	0.35	-0.8 $[-1.7, +0.0]$
Pilot ClassEval, Llama (53)	0.16	0.22	0.23	0.23	$+1.1$ $[+0.0, +2.3]$
Validation, Qwen (10)	0.29	0.37	0.37	0.38	$+1.0$ $[+0.0, +3.0]$
Validation, Llama (10)	0.27	0.43	0.50	0.50	$+7.0$ $[+0.0, +17.0]$

Table 5: Idealized single repairs (reference implementations; Qwen / Llama). Helped: states in which some single repair increases the number of components passing final tests. Downstream: share of the resulting fixes in components other than the repaired one. Next three columns: net increase in passing components from the top-ranked repair, divided by the best single repair's increase, conditional on a repair helping. Ties are averaged; states are averaged within task. The local signal is failure of the component's development tests. Intervals, unconditional gains, and candidate coverage are in Appendix C.9.

Workload	helped	downstream	local signal	degree × local	reach × local
Validation libraries	40/50 / 42/50	59% / 40%	0.49 / 0.43	0.61 / 0.81	0.83 / 0.86
Author suite	51/80 / 71/80	29% / 30%	0.57 / 0.58	0.79 / 0.83	0.85 / 0.88
ClassEval	197/265 / 228/265	21% / 25%	0.68 / 0.56	0.74 / 0.69	0.75 / 0.71
RepoClassBench	1/30 / 3/30	–	–	–	–

blies. Candidates come from fixed streams without replacement. Requests must fit the remaining budget at their worst-case token cost, and all policies share an acceptance rule and test limits (Appendix B.3). Budgets count generation tokens, with verifier calls recorded separately. The *degree* baseline uses one plus the number of direct dependents.

The *pilot* reuses 16 author-built libraries (90 functions) and 53 ClassEval classes (Du et al., 2023). Token costs are proxies because raw completions were not stored. The *validation set* has ten new author-written libraries of seven functions, frozen before candidate generation, with exact token counts and endpoints recorded before the first full run. Test limits changed after one validation result was inspected; Appendix B.3 gives the chronology and original-limit results. The primary endpoint is *whole-task success*: every component passes final tests in the assembled program. It measures practical utility outside the independent-error, partial-credit guarantee.

Allocation results. Table 4 separates the benefit of adaptation from the contribution of dependency weights. Adaptive reach matches uniform's whole-task success at the $3\times$ checkpoint using 0.42–0.59 of its tokens (point estimates; intervals and cost sensitivities in Appendix C.6). Most of this saving comes from stopping once a component passes, which the structure-blind adaptive baseline also does. Static structure-only allocation does not improve on uniform here.

The incremental effect of reach varies by setting. On the validation libraries the whole-task difference is $+7.0$ points $[+0.0, +17.0]$ for Llama and $+1.0$ points $[+0.0, +3.0]$ for Qwen, concentrated in two libraries and one library, respectively. The pilot includes a slightly negative ClassEval result for Qwen (-0.8 points $[-1.7, +0.0]$; -1.3 points $[-2.7, -0.0]$ in components passing). Direct degree performs similarly to reach on these code tasks. An exploratory analysis finds that the two policies often choose different components, but many runs stop drawing before the primary checkpoint. The proxy-weight corollary provides a way to assess this simpler alternative: its graph-only cost factor has median 1.10 on validation. This predicts a limited distinction under the homogeneous surrogate, consistent with the empirical comparison, though it does not bound these adaptive code runs (Appendix C.7).

Benefit of a single repair. We next replace each component of a fixed initial assembly by its reference implementation and rerun the final tests. This exploratory experiment measures the value of an ideal single repair, not achievable savings (Table 5). Across the validation libraries, author suite, and ClassEval, 21–59% of fail-to-pass changes occur in other components. Among states where some repair helps, reach times the local failure signal obtains 0.71–0.88 of the best net repair benefit, versus 0.43–0.68 for the signal alone. Averaged over all states, including those with no beneficial repair, these choices add 0.66–1.82 versus 0.58–0.76 passing components. Task-level intervals and regressions are reported in Appendix C.9.

Degree also improves target selection. The largest reach–degree difference is on the Qwen validation libraries (0.23 $[-0.03, 0.48]$); its interval includes zero. Elsewhere the differences are 0.01–0.05. The candidate pools help explain why better target selection gives smaller gains from resampling: for 10–58% of the best-target occurrences, no cached candidate passes its final tests even with reference dependencies. With more candidates such a target may still be repairable.

On six RepoClassBench classes (Deshpande et al., 2024) from one repository, all adaptive policies have identical outcomes and an ideal single repair helps in at most 3 of 30 states per model. With so few beneficial repairs, this exploratory study cannot distinguish repair priorities (Appendix C.8).

7.4 Executed propagation and boundary cases

The loss. On twelve synthetic trees at homogeneous $q = 0.72$, reach-weighted water-filling gains 1.22–2.61× under the corrupted-subtree loss and 0.48–0.97× under all-or-nothing loss, where uniform coincides with the numerical optimum (Theorem 5).

Executed propagation. On twelve executed arithmetic trees where a wrong parent answer changes the child's problem (Qwen2.5-7B, $B = 3n$; Appendix B.4), a deployable adaptive policy with structure-only caps corrupts 0.082 $[0.053, 0.118]$ of nodes with 2.5 of 3 votes per node, against 0.120 $[0.089, 0.156]$ for uniform voting. With the same adaptive stopping rule and uniform caps it corrupts 0.096 $[0.067, 0.126]$, and the runs are unpaired with variation of about 0.02. The incremental structural benefit is therefore suggestive, rather than established by this small study.

Floors. Tasks with large error floors give gains near one, as Proposition 8 predicts for the surrogate (Appendix B.5).

8 Discussion

When is structure useful? Dependency weights are motivated when correct intermediate work retains value, errors have unequal consequences, and additional effort can reduce them. Under homogeneous rates and continuous, unclipped allocations, two graph diagnostics quantify the benefit. If $\mathrm{AM}(w)/\mathrm{GM}(w)$ is near one, reach offers little surrogate improvement over uniform. If $\mathrm{AM}(w/d)/\mathrm{GM}(w/d)$ is near one, proxy weights d, such as degree, incur little additional surrogate loss (Corollary 7). Larger ratios motivate testing reach. Heterogeneous difficulty and adaptive feedback require empirical comparisons; these diagnostics are not deployment guarantees.

The repair diagnostic motivates future tests of targeted editing or stronger models for high-value components.

Limitations. The guarantees assume independent node errors and a given propagation graph. Arithmetic uses small, difficulty-selected pools and constructed dependencies. Code tests need not obey the propagation model exactly; whole-task success therefore tests practical behavior outside the partial-credit guarantee. The models are quantized 7–14B models, the code pilot uses approximate token costs, and the new validation libraries are small and author-written. Candidate streams are fixed, and reference replacement measures idealized repair value.

Conclusion. Downstream consequences give a quantitative basis for reliability budgets. Our rule combines a finite approximation guarantee, low median excess loss, and fast computation. Reach improves arithmetic allocation at tight budgets and idealized code repair targeting. The theory also characterizes the cost of simpler weights and the dependence on the loss, providing criteria for when dependency information is useful.

AI USE STATEMENT

We used Anthropic's Claude and OpenAI's GPT models to assist with experimental implementation of code. However the outcomes and the code itself have all been reviewed by the Authors. The models have also helped in polishing the writing including making the proofs look formal, the writing more well presented, and generating figures per the authors instructions. The authors did all of the writing (although this was revised by the models), developed the ideas, generated the theoretical results, designed the experiments, and interpreted the results themselves. The authors are responsible for the final text, claims, and artifacts.

REPRODUCIBILITY STATEMENT

The manuscript's numerical tables are generated from saved results by `make_numbers.py` and `gen_appendix.py`. Appendix B identifies experiment scripts, candidate caches, splits, budgets, and bootstrap units; it also records approximate costs and protocol deviations. The code comparisons use deterministic stream seeds. The exact tree dynamic program is checked against brute-force enumeration. Proofs are in Appendix A.

ETHICS STATEMENT

This work reallocates inference compute within existing systems and introduces no new data involving people. Model-generated programs were executed with timeouts in temporary working directories.

REFERENCES

Ehsan Aghazadeh, Ahmad Ghasemi, Hedyeh Beyhaghi, and Hossein Pishro-Nik. CGES: Confidence-guided early stopping for efficient and accurate self-consistency. *arXiv:2511.02603*, 2025.

Richard E. Barlow and Frank Proschan. *Mathematical Theory of Reliability*. Wiley, 1965.

Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, et al. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2024.

Shashi Borade, Baris Nakiboglu, and Lizhong Zheng. Unequal error protection: An information-theoretic perspective. *IEEE Transactions on Information Theory*, 55(12):5511–5539, 2009.

Chun-Hung Chen, Jianwu Lin, Enver Yücesan, and Stephen E. Chick. Simulation budget allocation for further enhancing the efficiency of ordinal optimization. *Discrete Event Dynamic Systems*, 10: 251–270, 2000.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more LLM calls all you need? towards scaling laws of compound inference systems. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Paula Cordero-Encinar and Andrew B. Duncan. Certified self-consistency: Statistical guarantees and test-time training for reliable reasoning in LLMs. *arXiv:2510.17472*, 2025.

Yingqian Cui, Zhenwei Dai, Pengfei He, Bing He, Hui Liu, Xianfeng Tang, Jingying Zeng, Suhang Wang, et al. Adaptive test-time reasoning via reward-guided dual-phase search. *arXiv:2509.25420*, 2025.

Ajinkya Deshpande, Anmol Agarwal, Shashank Shet, Arun Iyer, Aditya Kanade, Ramakrishna Bairi, and Suresh Parthasarathy. Class-level code generation from natural language using iterative, tool-enhanced reasoning over repository. *arXiv preprint arXiv:2405.01573*, 2024.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating LLMs on class-level code generation. *arXiv:2308.01861*, 2023.

Ji Huang, Wenbo Ma, and Zhi Zhou. Optimal bayesian stopping for efficient inference of consistent LLM answers. *arXiv:2602.05395*, 2026.

Way Kuo and V. Rajendra Prasad. An annotated overview of system-reliability optimization. *IEEE Transactions on Reliability*, 49(2):176–187, 2000.

Yixiao Li, Wei Deng, Jia Li, and Xiang Li. Spend less, reason better: Budget-aware value tree search for LLM agents. *arXiv:2603.12634*, 2026.

Junhong Lin, Xinyue Zeng, Jie Zhu, Song Wang, Julian Shun, Jun Wu, and Dawei Zhou. Plan and budget: Effective and efficient test-time scaling on reasoning large language models. *arXiv:2505.16122*, 2025.

Elliot Meyerson, Giuseppe Paolo, Roberto Dailey, Hormoz Shahrzad, Olivier Francon, Conor F. Hayes, Xin Qiu, Babak Hodjat, et al. Solving a million-step LLM task with zero errors. *arXiv:2511.09030*, 2025.

Adilson E. Motter and Ying-Cheng Lai. Cascade-based attacks on complex networks. *Physical Review E*, 66:065102, 2002.

Carlos N. Silla and Alex A. Freitas. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery*, 22:31–72, 2011.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling model parameters. In *International Conference on Learning Representations (ICLR)*, 2025.

Yashar Talebirad, Ali Parsaee, Vishwajeet Ohal, Amirhossein Nadiri, Csongor Szepesvari, Yash Mouje, and Eden Redman. Wisdom of the machines: Exploring collective intelligence in LLM crowds. In *Proceedings of the 25th International Conference on Autonomous Agents and Multiagent Systems*, pp. 2456–2463, 2026. doi: 10.65109/MTWR9974.

William E. Vesely, Francine F. Goldberg, Norman H. Roberts, and David F. Haasl. Fault tree handbook. Technical Report NUREG-0492, U.S. Nuclear Regulatory Commission, 1981.

Xiaoqiang Wan, Shengming Zhu, Jiale Cai, Guanzhong Chen, Xin Huang, Wei Zhou, and Maosong Sun. The shadow price of reasoning: Economic perspective on optimal budget allocation for LLMs. *arXiv:2606.03092*, 2026.

Fali Wang, Hui Liu, Zhenwei Dai, Jingying Zeng, Zhiwei Zhang, Zongyu Wu, Chen Luo, Zhen Li, Xianfeng Tang, Qi He, and Suhang Wang. AgentTTS: Large language model agent for test-time compute-optimal scaling strategy in complex tasks. *arXiv:2508.00890*, 2025a.

Xinglin Wang, Yiwei Li, Shaoxiong Feng, Peiwen Yuan, Yueqi Zhang, Jiayi Shi, Chuyi Tan, Boyuan Pan, Yao Hu, and Kan Li. Every rollout counts: Optimal resource allocation for efficient test-time scaling. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b. arXiv:2506.15707.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Fengbin Zhu, Wenqiang Lei, Youcheng Huang, Chao Wang, Shuo Zhang, Jiancheng Lv, Fuli Feng, and Tat-Seng Chua. TAT-QA: A question answering benchmark on a hybrid of tabular and textual content in finance. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics*, 2021.

A PROOFS

A.1 REMARK 1

Let $X \sim \mathrm{Bin}(2k-1, q)$ count correct votes among the first $2k-1$ and let Y be the $2k$-th vote. With $2k-1$ votes the majority errs iff $X \le k-1$. With $2k$ votes it errs iff $X + Y < k$, and errs with probability $\frac{1}{2}$ iff $X + Y = k$. Conditioning on X: both rules err for $X \le k-2$ and neither for $X \ge k+1$; for $X = k-1$ the $2k$-rule errs with probability $(1-q) + q/2$ against 1; for $X = k$ with probability $(1-q)/2$ against 0. The difference is $-\frac{q}{2}P(X = k-1) + \frac{1-q}{2}P(X = k) = 0$, since $\binom{2k-1}{k} = \binom{2k-1}{k-1}$ gives the cancellation for $0 < q < 1$; the endpoints follow directly. □

Rounding used in the voting experiments. Every node receives one vote; the $\lfloor (B-n)/2 \rfloor$ available pairs are assigned by largest remainder of $(b_v - 1)/2$; if $B - n$ is odd the leftover vote goes to the node with the largest b_v. Uniform at $B = kn$ with k odd gives exactly k votes per node, so the restriction to odd counts constrains only the non-uniform policies. Verifier selection uses ordinary largest-remainder rounding.

A.2 LEMMA 1 AND COROLLARY 2

Node v is corrupted iff some $u \preceq v$ errs. The union bound gives $P(v \text{ corrupted}) \le S_v$, and summing over v counts ε_u once for each $v \in \mathrm{subtree}(u)$, that is w_u times, so $\mathcal{L} \le L$. Under independence, $P(v \text{ corrupted}) = 1 - \prod_{u \preceq v}(1 - \varepsilon_u) \ge 1 - e^{-S_v}$ because $1 - x \le e^{-x}$. Since $\psi\prime(g) = ((g+1)e^{-g} - 1)/g^2 \le 0$ for $g > 0$, $1 - e^{-S_v} = S_v\psi(S_v) \ge S_v\psi(\gamma_{\mathrm{path}})$; summing with the same weight counting gives $\psi(\gamma_{\mathrm{path}})L \le \mathcal{L}$. Finally $e^g \ge 1 + g$ gives $e^{-g} \le 1/(1+g)$, hence $1 - e^{-g} \ge g/(1+g)$ and $\psi(g) \ge 1/(1+g)$. For the corollary, $\mathcal{L}(b^\star) \le L(b^\star) \le L(b^{\mathrm{true}}) \le \mathcal{L}(b^{\mathrm{true}})/\psi(\gamma_{\mathrm{path}}(b^{\mathrm{true}}))$. □

A.3 THEOREM 3 AND COROLLARY 4

L is a sum of strictly convex functions of separate variables, so the KKT conditions are necessary and sufficient: $c_v w_v e^{-c_v b_v} = \lambda$ on the active set and $c_v w_v \le \lambda$ off it, giving the stated form; $\sum_v b_v^\star(\lambda)$ is strictly decreasing where positive, so λ and $b^\star$ are unique. With $c_v \equiv c$ and no clipping, the budget fixes $c^{-1}\ln(c/\lambda) = B/n - c^{-1}\ln \mathrm{GM}(w)$; at the optimum $e^{-cb_v^\star} = \lambda/(cw_v)$, so $L(b^\star) = n\lambda/c = ne^{-cB/n}\mathrm{GM}(w)$, while $L(\mathrm{unif}) = ne^{-cB/n}\mathrm{AM}(w)$. Equality in AM–GM needs all w_v equal, and a rooted tree has $w_{\mathrm{root}} = n$ and $w_{\mathrm{leaf}} = 1$. □

A.4 APPROXIMATE WEIGHTS

Corollary 7 (Cost of proxy weights)**.** *Let $c_v \equiv c$, let d be positive proxy weights, and let $b(d)$ be the water-filling allocation computed from d at budget B. If neither $b(d)$ nor $b(w)$ clips at zero, then $L_w(b(d))/L_w(b(w)) = \mathrm{AM}(w/d)/\mathrm{GM}(w/d)$, where $L_w(b) = \sum_v w_v e^{-cb_v}$.*

Proof. Unclipped water-filling on d gives $b_v(d) = B/n + c^{-1}(\ln d_v - \overline{\ln d})$, so $e^{-cb_v(d)} = e^{-cB/n}\mathrm{GM}(d)/d_v$ and $L_w(b(d)) = e^{-cB/n}\mathrm{GM}(d)\sum_v w_v/d_v = ne^{-cB/n}\mathrm{GM}(d)\mathrm{AM}(w/d)$. By Corollary 4, $L_w(b(w)) = ne^{-cB/n}\mathrm{GM}(w)$, and $\mathrm{GM}(w)/\mathrm{GM}(d) = \mathrm{GM}(w/d)$. □

Uniform allocation is the case $d \equiv 1$, which recovers Corollary 4. The corollary covers neither integer or clipped allocations, heterogeneous rates, feedback-adaptive policies, nor whole-task success; Appendix C.7 therefore also reports the exact integer comparison at the tested budgets.

A.5 ERROR FLOORS

Proposition 8 (Floor-aware law)**.** *With positive floor mass $F = \sum_v w_v \varepsilon_{\infty,v}$ and reducible part $R(b) = \sum_v w_v A_v e^{-c_v b_v}$, the minimizer of $F + R$ is the water-filling of Theorem 3 on the reducible coefficients $w_v A_v$, and $\frac{F+R(\mathrm{unif})}{F+R(b^\star)} = \frac{1+\rho}{1+\rho/g_{\mathrm{red}}}$ with $\rho = R(\mathrm{unif})/F$ and $g_{\mathrm{red}} = R(\mathrm{unif})/R(b^\star)$.*

For $F = 0$ the gain is g_{red}. F does not depend on b, so the minimizer of $F + R$ minimizes R, which is Theorem 3 with w_v replaced by $w_v A_v$. Dividing $(F + R_u)/(F + R_\star)$ by F gives the ratio, increasing in ρ with limits 1 and g_{red}. □

A.6 Token costs and DAGs

Proposition 9 (Cost-weighted water-filling). *Minimizing $\sum_v w_v A_v e^{-c_v b_v}$ subject to $\sum_v \kappa_v b_v \le T$, $b \ge 0$ gives $b_v^\star = \max(0, c_v^{-1}[\ln(c_v w_v A_v/\kappa_v) - \ln\lambda])$; when the costs of active nodes differ, the cost-blind allocation violates the KKT conditions and is strictly suboptimal.*

Proposition 10 (DAGs). *On a DAG in which an error at v corrupts every node that depends on it, set $w_v = |\mathrm{reach}(v)|$ and let γ_{path} be the largest error mass over an ancestor set $\{u : v \in \mathrm{reach}(u)\}$. Then Lemma 1, Theorem 3, and Propositions 8 and 9 hold unchanged, and under Corollary 4's conditions the gain is $\mathrm{AM}(w)/\mathrm{GM}(w) \ge 1$ with equality iff there are no edges.*

Both proofs repeat the arguments above. On a DAG, corruption of a node is a single event however many paths reach it, so the union bound and the exchange of summation count ε_v exactly $|\mathrm{reach}(v)|$ times. An edge $u \to v$ implies $\mathrm{reach}(u) \supsetneq \mathrm{reach}(v)$ by acyclicity, which makes AM–GM strict.

A.7 Theorem 5

Part (1) is Theorem 3. For (2), minimizing $\mathcal{L}_{\mathrm{aon}}$ maximizes $\Phi(b) = \sum_v \log(1 - e^{-c_v b_v})$. Any allocation with some $b_v = 0$ has $\Phi = -\infty$, so the optimum is interior; each summand is strictly concave, so the maximizer is unique and satisfies $c_v/(e^{c_v b_v} - 1) = \lambda$, which involves c_v and λ only. With homogeneous rates all b_v coincide. □

A.8 Plug-in stability

Lemma 11 (Plug-in stability). *Let $\mathcal{B} \subseteq \mathbb{R}_+^n$ be a nonempty compact convex set of allocations and $b^\star(c) = \arg\min_{b \in \mathcal{B}} L(c, b)$ with $L(c, b) = \sum_v w_v e^{-c_v b_v}$. If $\hat{c}_v \ge c_v/2$ for all v, then $0 \le L(c, b^\star(\hat{c})) - L(c, b^\star(c)) \le \frac{4}{e}\big(\sum_v w_v/c_v\big) \max_v |\hat{c}_v - c_v|$.*

Proof. Optimality of $\hat{b} = b^\star(\hat{c})$ for $\hat{c}$ over $\mathcal{B}$ gives $L(c, \hat{b}) - L(c, b^\star) \le 2 \max_{b \in \{\hat{b}, b^\star\}} |L(c, b) - L(\hat{c}, b)|$. For fixed b, the mean value theorem gives $|L(c, b) - L(\hat{c}, b)| \le \sum_v w_v b_v e^{-\tilde{c}_v b_v} |c_v - \hat{c}_v|$ with $\tilde{c}_v \ge c_v/2$, and $\sup_{t \ge 0} t e^{-(c_v/2)t} = 2/(e c_v)$. □

With a single pooled rate $\bar{c}$ satisfying $\bar{c} \ge c_v/2$ for all v, the lemma bounds the excess of the structure-only allocation by $\frac{4}{e}(\sum_v w_v/c_v) \max_v |\bar{c} - c_v|$, which vanishes only when the nominal rate approaches the common true rate; homogeneous true rates alone are not enough if $\bar{c}$ differs from them.

A.9 Proposition 6

With exact homogeneous rates and water-filling unclipped at budget $B - mn$, Corollary 4 gives $L^\star(B - mn) = n\mathrm{GM}(w) e^{-c(B-mn)/n} = L^\star(B) e^{cm}$, and $L_{\mathrm{unif}}(B) = n\mathrm{AM}(w) e^{-cB/n}$. The gain $L_{\mathrm{unif}}(B)/L^\star(B - mn) = (\mathrm{AM}/\mathrm{GM}) e^{-cm}$ is above one iff $m < c^{-1} \ln(\mathrm{AM}/\mathrm{GM})$. □

A.10 Reused probes

Proposition 12 (Reused-probe surrogate bound). *Let $q_v > \frac{1}{2}$, $c_v = 2(q_v - \frac{1}{2})^2$, and $c_{\min} = \min_v c_v$. Let $\hat{q}_v$ be the empirical accuracy of m i.i.d. labeled Bernoulli probes per node, with $\hat{c}_v = 2(\hat{q}_v - \frac{1}{2})^2$, and suppose $B \ge mn$. The probes are counted as votes. Let $\hat{b}$ water-fill the full budget on $\hat{c}$ subject to $b_v \ge m$. If $\eta = \sqrt{\ln(2n/\delta)/(2m)} \le c_{\min}/4$, then with probability at least $1 - \delta$, $L(c, \hat{b}) - L_m^\star(c, B) \le \frac{8}{e}\big(\sum_v w_v/c_v\big)\eta$, where $L_m^\star$ is the optimum over allocations with $b_v \ge m$.*

By Hoeffding's inequality and a union bound, $\max_v |\hat{q}_v - q_v| \le \eta$ with probability at least $1 - \delta$. Since $|2(\hat{q} - \frac{1}{2})^2 - 2(q - \frac{1}{2})^2| = 2|\hat{q} - q||\hat{q} + q - 1| \le 2|\hat{q} - q|$, we have $\max_v |\hat{c}_v - c_v| \le$

$2\eta \le c_{\min}/2$, so $\hat{c}_v \ge c_v/2$. Applying Lemma 11 on the convex set $\{b : \sum_v b_v = B,\, b_v \ge m\}$, which contains both $\hat{b}$ and the restricted optimum, gives the bound. Two limitations follow from the statement. The quantity bounded is the surrogate evaluated at the data-dependent allocation, not the realized error of voting with the reused samples. The condition $\eta \le c_{\min}/4$ with $c_{\min} \le \frac{1}{2}$ requires $m \ge 32\ln(2n/\delta)$. □

A.11 Exact tree dynamic program

Proposition 13 (Certified optimum)**.** *On a rooted tree with independent node errors and arbitrary per-node error tables $\varepsilon_v(b)$, the minimum of $\mathcal{L}$ over integer allocations with $b_v \in \mathcal{A}$ (for example, odd counts) and $\sum_v b_v \le B$ is computed exactly by the recursion $F_v(k) = \max_{b\in\mathcal{A},\, b\le k} \big(1-\varepsilon_v(b)\big)\big(1+G_v(k-b)\big)$, where $G_v(r) = \max_{\sum_c k_c \le r} \sum_c F_c(k_c)$ combines the children's budgets, and $\min \mathcal{L} = n - F_{\mathrm{root}}(B)$. Set $G_v(r) = 0$ for a leaf and $F_v(k) = -\infty$ when no allocation is feasible.*

Proof. For a subtree rooted at v whose ancestors all survive with probability s, its contribution to the loss is $\sum_{u\in\mathrm{subtree}(v)}(1 - s\,P_u) = |\mathrm{subtree}(v)| - s\sum_u P_u$, where P_u is the product of $(1-\varepsilon_x)$ along the path from v to u. The minimizing allocation within the subtree therefore maximizes $\sum_u P_u$ regardless of s, and $\sum_u P_u = (1-\varepsilon_v(b_v))(1+\sum_c \sum_{u\in\mathrm{subtree}(c)} P_u^{(c)})$, which separates across children given their budgets. Induction over the tree gives the recursion. □

The recursion costs $O(nB^2)$ time. We verified it against brute-force enumeration on 40 random trees with up to six nodes; the maximum absolute difference was 9×10^{-16}.

B Experimental protocols

The deployable reach-weighted policy. Algorithm 1 states the sequential policy used in all experiments. It sees only the graph, its own draws, and a fixed prior; the blind, degree, and shuffled variants change only the weight w_v.

Algorithm 1 Reach-weighted adaptive allocation (one task)

Require: dependency graph with reach w_v; budget B; verifier or vote-agreement estimate
1: draw one sample per node; $b_v \leftarrow 1$; spent $\leftarrow n$ (or its token cost)
2: **while** a request fits under B and some node is unresolved **do**
3: for each unresolved node v: $g_v \leftarrow w_v \cdot \big[\hat{\varepsilon}_v(b_v) - \hat{\varepsilon}_v(b_v{+}1)\big]$ ▷ code: $g_v \leftarrow w_v/(b_v{+}1)$
4: $u \leftarrow \arg\max_v g_v$; draw one more sample for u; $b_u \leftarrow b_u + 1$; charge its cost
5: update u's selected answer (vote) or accept the candidate iff it passes more development tests
6: **end while**
7: **return** the selected answer or candidate of every node

Models and sampling. All model outputs come from Llama-3.1-8B-Instruct, Qwen2.5-7B-Instruct, and Qwen2.5-14B-Instruct, 4-bit quantized, served by `mlx-lm` on one Apple-silicon machine with batched sampling. Arithmetic answers are sampled at temperature 1.0 and code at 0.8. Seeds are deterministic functions of the problem and sample index.

Bootstrap units. Intervals are 95% percentile bootstrap intervals over the independent unit of each experiment: random tree assignments (arithmetic), runs (end-to-end), and libraries or classes (code). Gains are ratios of pooled losses. Paired comparisons resample the unit and recompute both policies on the same resample.

B.1 Synthetic trees

Seventeen shapes are used: balanced binary trees of depth 3–6, ternary trees of depth 3–4, a quaternary tree of depth 3, stars on 16, 32, and 64 nodes, chains of 8, 16, and 32, and random recursive

trees on 15, 31, 63, and 127 nodes. Node accuracies are homogeneous, $q \in \{0.62, 0.68, 0.72, 0.78\}$, and vote errors are exact binomial tails with fair ties. The certified optimum is the dynamic program of Proposition 13 over odd counts. Script: `exp20_synthetic_parity.py`. On random trees at $B = 3n$, water-filling takes 0.1–5 ms against 4 ms to 6.8 s for the exact program as tree size grows from 15 to 511, a speedup from $34\times$ to about $1{,}300\times$.

Parity. At two votes per node, uniform allocation has exactly the loss of one vote per node on every configuration. A structure-blind control that gives a random half of the nodes three votes and the rest one gains $1.03\times$ ($q = 0.62$) and $1.10\times$ ($q = 0.72$) over two-vote uniform solely because of parity. Water-filling with odd rounding gains $1.19\times$ and $1.44\times$.

B.2 Arithmetic sub-problem trees

For each model we generated 192 new multiplication problems using operand lengths chosen to vary difficulty (Llama: 2×2 and 3×1 digits; Qwen-7B: 2×2, 3×2, 4×2; Qwen-14B: 4×2 and 3×3) and drew 8 raw integer *calibration* answers per problem. A fixed seed then selects 16 *calibration problems*, which never appear in a tree. They supply the only pool-level information the deployable policies use: the mean modal-answer share $\bar{m}$, which sets the prior $\mathrm{Beta}(4\bar{m}, 4(1-\bar{m}))$ on a node's agreement rate, and the nominal rate $2(\bar{m} - \frac{1}{2})^2$ for structure-only allocation. No labels are used for either. Among the remaining problems, those with between 3 and 7 correct calibration answers form the band, up to 48 per model, and the analysis keeps those that are votable on their calibration answers (at least 5 of 8 correct). Each banded problem then received 16 new *evaluation* answers. The oracle policies' curves are the exact binomial tail at the Laplace-smoothed calibration accuracy, fitted by an exponential. Every scoring curve (the plurality error when b votes are drawn with replacement from the evaluation answers, estimated with 1,500 resamples) and every online draw uses evaluation answers only. Thirteen topologies (5–63 nodes) are populated by drawing problems at random, independently of position, with 40 assignments per topology. Static policies are scored by their expected loss under the evaluation curves, and sequential policies by the realized plurality outcomes of their own draws, averaged over three draws per assignment. The label-free adaptive policies estimate a node's accuracy by its modal-answer share under the prior and spend votes in pairs, keeping every count odd; the adaptive warm start draws three votes per node and then allocates the rest with the same rule. The first fresh pool (Table 7) used the same pipeline with 64 problems per model, 24 answers each (answers 0–7 for calibration, 8–23 for evaluation), and no band. Scripts: `exp24_arith_collect_band.py`, `exp24_arith_analyse.py`.

B.3 Code benchmarks

Author-built suite. Sixteen libraries with 90 interdependent functions. Each has a reference implementation, a dependency DAG, a one-line specification per function, and unit tests; every reference passes, and replacing a function with one that raises makes its dependents fail. Twelve candidates per function were drawn from its specification and the specifications of its dependencies. A held-out battery of 30 randomized inputs per function is checked against the reference on the assembled program.

ClassEval. Eligibility was fixed before sampling: every ClassEval (Du et al., 2023) class that declares at least one intra-class method dependency (60 classes), minus classes whose reference solution fails its own tests in our harness (4: one class's tests depend on the current date, and three import libraries unavailable in our environment). This leaves 56 classes and 243 methods. Each method's test functions, in order, are split: the first half are *development* tests, the rest *final* tests; 3 methods have a single test, used for both, and a sensitivity analysis excludes their classes (ClassEval_0, _62, _89). Eight candidates per method were drawn from a prompt that gives the class description, imports and constructor, one-line summaries of the other methods, and the target method's signature and docstring. Reach weights come from the benchmark's declared `method_dependencies`.

Realistic-verifier protocol. Every policy starts from the same initial assembly (the first candidate of each component's stream) and is charged its tokens; this initial state is reported as the first point of every curve. Candidate streams are the cached candidates in a random order drawn from a stable SHA-256 seed per task, model, and repetition, shared by all policies and consumed without replacement; exhaustion is recorded. Development feedback runs the component's development tests on

the policy's current assembly with no reference code. Final tests run in separate processes, from separate specifications, only on saved states after the policy has finished. Every test run has the same explicit limits: 5 CPU-seconds, 5 s of requested sleep (sleep calls do not wait but are counted, which stops code that loops on a timer), and a 20 s wall-clock limit. Runs stopped by a limit fail and are counted. All policies use one acceptance rule: a drawn candidate replaces the current one iff it passes strictly more of the component's development tests in the current assembly (ties keep the current candidate). A request is issued only if the spend so far plus the prompt tokens plus the maximum completion length fits under the cap of four times the initialization cost, so no policy looks ahead at a candidate's cost; the actual cost is charged after generation. The state at checkpoint 1.5, 2, 3, or 4 times the initialization cost is the last state whose actual spend is within it. These lower checkpoints are retrospective points on a trajectory reserved against the $4\times$ cap, not separate runs with admission against each lower cap. Static policies convert the cap into a candidate count with the mean candidate cost of the *other* tasks, plan per-component counts (uniform, proportional to reach, or exact integer water-filling on reach at nominal rate $\ln 2$, by greedy marginal gains with $1 \le b_v \le K$), and draw them in round-robin rounds in dependency order. Sequential policies choose among components that fail a development test and have candidates left: blind (fewest draws), reach ($\max w_v/(d_v+2)$), degree (the same with $1+$ the number of direct dependents), or shuffled reach (weights permuted within the task). There are 10 repetitions per task, the task is the statistical unit, and paired bootstrap intervals (2000 resamples) resample tasks. Token ratios are computed at fixed targets (uniform's pooled quality at $2\times$ and $3\times$ the initialization tokens) by interpolating the pooled curves and bootstrapping over tasks; resamples in which a policy never reaches the target are counted, never extrapolated. *Costs in the pilot are a proxy*: raw completions were not stored for these older candidates, so a candidate costs its chat-templated prompt tokens plus its extracted-code tokens plus 12 (450/460 if extraction failed). We report two sensitivity variants: a $+80$ overhead and a flat per-candidate cost. The pilot excludes the three ClassEval classes whose single test served as both development and final test. Scripts: `realistic2/engine.py`, `tasks.py`, `run_pilot.py`, `analyse2.py`.

Validation set. Ten new libraries of seven functions each (text statistics, linear algebra, rational arithmetic, graph distances, Z-function string matching, order statistics, polynomial integration, bit manipulation, calendar arithmetic, and an inventory ledger) were written by the authors, with reference implementations, unit tests, and random-input generators, and frozen before any candidate was generated. A check script confirms that every reference passes its tests, that breaking any dependency fails every dependent's tests, and that references are deterministic on the final inputs. In 5 of the 10 libraries no function is ranked first by both reach and degree. Candidates (10 per function, temperature 0.8, at most 460 tokens) were generated fresh for both models, and the raw completions and exact prompt and completion token counts were stored; costs are exact. The final test is a 30-input differential battery per function run in the assembled library; the development test is the function's unit tests with its current generated dependencies. Scripts: `realistic2/newlibs.py`, `verify_newlibs.py`, `gen_newlibs.py`, `run_confirm.py`.

Chronology and deviations. The endpoints, contrasts, and targets were written down before the first full policy run on the validation libraries. Before that, a two-repetition smoke test of the runner on one library displayed only the shared initial state, and a diagnostic ran each Qwen candidate with reference dependencies on the final inputs to check the harness; it showed several functions with no correct candidate, which were left unchanged. The first Qwen validation run used a 5 s wall-clock limit while other jobs loaded the machine; it recorded 49 development-test timeouts, and a low-load diagnostic showed they had rejected correct code. After that result had been inspected, the limit was changed to the CPU-time limit described above and every run was repeated; the wall-clock results are reported in Table 14. A 60 s and then 20 s wall-clock limit and the sleep budget were introduced because some ClassEval candidates loop on timers; no validation or author-suite candidate calls `sleep`, so those results do not depend on this change. The bootstrap for token ratios originally dropped resamples in which the initial state already met the target; they now count as a ratio of 1. The full log accompanies the released code.

B.4 END-TO-END TREES

Each node computes $x_v \times m_v$ for a fixed two-digit operand m_v and the input x_v it receives. A child's input is the last two digits of its parent's plurality answer, mapped into 10–99. A node is counted as corrupted when its answer differs from the value it would have produced on the correct input. This is an executed-output definition: a wrong parent answer that happens to map to the correct child input does not corrupt the child, so an error does not always propagate to its descendants, unlike in the corrupted-subtree loss. Twelve trees of 4–15 nodes are used. Static policies other than uniform, weight-proportional, and structure-only use per-node curves fitted from ten cached answers to each node's own problem, which are not charged to the budget; they are oracle references. The label-free adaptive policies draw votes one at a time, stop a node once its leading answer is two votes ahead, and allocate unused votes to later nodes. Their per-node caps come from uniform allocation, from structure-only water-filling at the nominal rate 0.5 (no calibration), or from the oracle water-filling. Every answer a run needs is sampled live if not cached, so no run is skipped. Script: `exp22_e2e_parity.py`.

Results. All policies are capped at 9 votes per node, and counts report actual draws. Under chain-of-thought answering at $B = 3n$, uniform corrupts 0.120 $[0.089, 0.156]$ of nodes. The adaptive policy with structure-only caps corrupts 0.082 $[0.053, 0.118]$ using 2.5 votes per node; uniform caps give 0.096 $[0.067, 0.126]$ and oracle caps give 0.054 $[0.030, 0.080]$. Runs are unpaired, and a repeat with fresh draws moved individual adaptive results by about 0.02, as large as their differences. We therefore do not rank the adaptive variants. Under direct answering, uniform corrupts 0.197 $[0.168, 0.226]$ at $B = 3n$ and 0.200 $[0.169, 0.227]$ at $B = 5n$, with no clear policy advantage (Appendix C.4).

B.5 TASKS WITH LARGE ERROR FLOORS

When most of the invalidated work comes from nodes that resampling cannot fix, Proposition 8 predicts surrogate gains near one. Three studies are consistent with this result. The arithmetic evaluation problems taken without difficulty selection are dominated by problems the models almost never answer correctly, and every policy there is within 2% of uniform (Appendix C.1). On CIFAR-100's superclass-to-fine hierarchy with a frozen CNN ensemble, the fitted floors give $\rho = 0.22$, and Proposition 8 gives $1.07\times$ against $1.10\times$ measured; the floor was fitted on the same vote curves, so this is a consistency check rather than an advance prediction. On six CPython modules regenerated from their docstrings, water-filling gains $1.06\times$ $[1.01, 1.17]$, and four of the six modules are dominated by functions with no correct regeneration in the sampled candidates. The gains in these settings are small but nonzero; the optimal allocation still depends on the graph.

C FULL RESULTS

C.1 ARITHMETIC: FULL RESULTS

Table 6 gives every budget for the difficulty-banded pools of §7.2. Table 7 gives the first fresh pool, drawn without a difficulty band. Most of its problems are either solved on every calibration answer or almost never, so the votable subset has little error left to allocate, and the unfiltered set is dominated by problems that voting rarely solves over the tested budgets. Entries marked – are ratios of losses that are numerically zero (the votable Qwen2.5-14B problems were answered correctly on every evaluation answer).

C.2 SENSITIVITY TO CALIBRATION AND VOTE PARITY

On fixed instances of the earlier arithmetic pools (15 correctness flags per problem, votable problems, 13 topologies $\times$ 40 assignments), we vary three factors: the budget and rounding, the accuracy model, and the curve used to allocate. The evaluation curve is the exact binomial tail at the accuracy model's q. Treating a problem correct on all cached samples as error-free (raw k/K) inflates the gain far more than any other factor.

Table 6: Difficulty-banded arithmetic pools: gain over uniform, all budgets. The warm start draws 3 votes per node first and is defined only for $B \geq 5n$.

Pool	B	weight	struct.	warm st.	ad. blind	ad. reach	diff. (or.)	wf (or.)	reach/blind
Llama-3.1-8B, votable (31)	$3n$	1.13	1.13	–	0.99	1.12	0.99	1.13	1.13 [1.09,1.16]
	$5n$	1.08	1.09	1.10	0.94	1.10	0.99	1.09	1.17 [1.12,1.22]
	$7n$	1.05	1.05	1.04	0.97	1.03	0.97	1.06	1.06 [1.02,1.11]
	$9n$	1.05	1.07	1.04	1.04	1.05	0.98	1.06	1.02 [0.98,1.05]
Qwen2.5-7B, votable (17)	$3n$	1.07	1.07	–	1.03	1.11	0.97	1.08	1.08 [1.05,1.11]
	$5n$	1.03	1.03	1.05	1.01	1.07	0.96	1.04	1.06 [1.03,1.10]
	$7n$	1.02	1.02	1.07	1.02	1.06	0.97	1.03	1.04 [1.00,1.07]
	$9n$	1.01	1.02	1.06	1.04	1.04	0.97	1.02	1.00 [0.97,1.03]
Qwen2.5-14B, votable (17)	$3n$	1.12	1.12	–	1.02	1.14	0.98	1.12	1.12 [1.08,1.16]
	$5n$	1.06	1.08	1.10	1.03	1.10	0.99	1.08	1.06 [1.03,1.10]
	$7n$	1.04	1.06	1.05	1.01	1.07	0.98	1.05	1.06 [1.02,1.11]
	$9n$	1.00	1.04	1.04	0.99	1.01	0.98	1.03	1.02 [0.99,1.06]

Table 7: First fresh arithmetic pool, without a difficulty band.

Pool	B	weight	struct.	warm st.	ad. blind	ad. reach	diff. (or.)	wf (or.)	reach/blind
Llama-3.1-8B, votable (28)	$3n$	1.26	1.22	–	0.96	1.76	1.37	1.70	1.84 [1.54,2.19]
	$5n$	1.32	1.30	1.60	1.42	1.99	2.33	2.58	1.40 [1.04,1.90]
	$7n$	1.34	1.43	1.72	1.57	1.37	3.80	4.18	0.87 [0.64,1.20]
	$9n$	1.49	1.56	1.38	1.25	1.53	5.59	6.44	1.23 [0.91,1.63]
Llama-3.1-8B, all (48)	$3n$	1.01	1.01	–	1.00	1.00	1.01	1.01	1.00 [0.99,1.01]
	$5n$	1.01	1.01	1.00	1.00	1.00	1.00	1.00	1.00 [0.99,1.01]
	$7n$	1.00	1.00	1.00	1.01	1.00	1.00	1.00	1.00 [0.99,1.01]
	$9n$	1.00	1.01	1.00	1.00	0.99	1.00	1.00	0.99 [0.98,1.00]
Qwen2.5-7B, votable (28)	$3n$	1.09	1.01	–	1.00	1.03	1.04	1.11	1.04 [0.95,1.14]
	$5n$	1.04	0.97	0.98	0.94	1.03	1.04	1.07	1.10 [1.01,1.20]
	$7n$	1.01	0.94	0.99	0.99	0.96	1.02	1.03	0.98 [0.91,1.05]
	$9n$	1.00	0.93	0.93	0.93	0.88	1.00	1.01	0.94 [0.87,1.01]
Qwen2.5-7B, all (48)	$3n$	1.00	1.00	–	1.00	1.00	1.00	1.01	1.00 [1.00,1.01]
	$5n$	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.00 [0.99,1.00]
	$7n$	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.00 [1.00,1.00]
	$9n$	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.00 [1.00,1.01]
Qwen2.5-14B, votable (17)	$3n$	0.99	0.21	–	1.70	0.86	1.00	3.72	0.51 [0.20,1.02]
	$5n$	0.28	0.03	0.25	0.29	0.55	1.00	3.80	1.90 [0.35,5.73]
	$7n$	0.47	0.03	2.76	0.46	3.69	–	44.58	–
	$9n$	–	–	–	–	–	–	–	–
Qwen2.5-14B, all (48)	$3n$	1.00	1.00	–	1.00	1.00	1.00	1.00	1.00 [1.00,1.00]
	$5n$	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00 [1.00,1.00]
	$7n$	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00 [1.00,1.00]
	$9n$	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00 [1.00,1.00]

Table 8: Gain of oracle water-filling over uniform on fixed instances.

Model	budget, rounding	raw, exp. fit	raw, exact	Laplace, exp. fit	Laplace, exact
Llama-3.1-8B	$2n$, even allowed	2.12	1.10	1.69	1.32
	$2n$, odd	2.09	2.34	1.60	1.61
	$3n$, odd	2.94	3.17	1.60	1.62
Qwen2.5-14B	$2n$, even allowed	1.91	1.05	1.62	1.27
	$2n$, odd	1.84	1.97	1.52	1.53
	$3n$, odd	2.19	2.30	1.48	1.49
Qwen2.5-7B	$2n$, even allowed	2.19	1.20	1.68	1.42
	$2n$, odd	2.12	2.53	1.62	1.64
	$3n$, odd	3.25	3.79	1.63	1.65

C.3 CODE UNDER AN IDEALIZED COMPONENT VERIFIER

These results come from an earlier protocol in which a candidate's selection tests were run with the correct reference implementations of every other component, which a deployed system does not have. All policies shared this advantage, so within-protocol comparisons are valid, but the protocol overstates deployable feedback; the main text uses the realistic verifier instead. The author-suite numbers here come from a run whose candidate draws were seeded with Python's salted string hash, so they reproduce only up to that randomness; the ClassEval numbers were rerun with stable seeds.

Table 9: Executed code: gain over uniform by the selection tests and by the held-out tests, library-level pass rate, and total corrupted functions (held-out), summed over libraries or classes and averaged over repetitions.

Setting	metric	unif.	weight	struct.	1-probe	ad. blind	ad. reach	diff. (or.)	wf (or.)
Suite, Qwen-7B, $2n$	selection tests	1.00	0.98	0.97	1.62	1.67	1.79	1.09	1.30
	held-out tests	1.00	1.03	1.03	1.33	1.33	1.41	1.10	1.21
	library pass (held-out)	0.45	0.44	0.43	0.50	0.51	0.52	0.49	0.48
	corrupted, held-out total	21.5	20.8	20.9	16.1	16.1	15.2	19.5	17.8
Suite, Qwen-7B, $3n$	selection tests	1.00	0.95	0.95	1.39	1.65	1.67	1.14	1.26
	held-out tests	1.00	0.99	1.00	1.22	1.28	1.30	1.07	1.16
	library pass (held-out)	0.51	0.49	0.49	0.55	0.55	0.55	0.53	0.53
	corrupted, held-out total	17.8	18.0	17.9	14.6	13.9	13.8	16.6	15.4
Suite, Llama-8B, $2n$	selection tests	1.00	0.95	0.93	1.39	1.52	1.53	1.10	1.13
	held-out tests	1.00	0.99	0.99	1.24	1.28	1.30	1.05	1.11
	library pass (held-out)	0.24	0.22	0.20	0.34	0.35	0.36	0.26	0.27
	corrupted, held-out total	34.1	34.3	34.5	27.6	26.6	26.2	32.4	30.7
Suite, Llama-8B, $3n$	selection tests	1.00	0.95	0.95	1.34	1.68	1.67	1.16	1.19
	held-out tests	1.00	1.00	1.00	1.18	1.29	1.30	1.10	1.11
	library pass (held-out)	0.28	0.27	0.27	0.36	0.41	0.41	0.33	0.33
	corrupted, held-out total	30.7	30.7	30.7	26.0	23.8	23.7	27.8	27.5
ClassEval, Qwen2.5-7B, $2n$	selection tests	1.00	0.99	0.97	1.10	1.12	1.11	1.01	1.05
	held-out tests	1.00	1.00	0.98	1.06	1.07	1.07	1.01	1.03
	library pass (held-out)	0.25	0.24	0.23	0.29	0.30	0.29	0.27	0.26
	corrupted, held-out total	89.9	90.3	91.6	84.9	84.3	84.2	89.1	87.6
ClassEval, Qwen2.5-7B, $3n$	selection tests	1.00	0.98	0.99	1.10	1.13	1.11	1.01	1.06
	held-out tests	1.00	0.99	0.99	1.06	1.07	1.06	1.00	1.03
	library pass (held-out)	0.27	0.26	0.26	0.30	0.31	0.31	0.29	0.29
	corrupted, held-out total	86.8	87.3	87.4	82.1	81.1	81.6	86.4	83.9
ClassEval, Llama-3.1-8B, $2n$	selection tests	1.00	0.99	0.99	1.08	1.09	1.10	0.96	1.04
	held-out tests	1.00	0.99	0.99	1.06	1.06	1.07	0.97	1.02
	library pass (held-out)	0.10	0.10	0.10	0.14	0.15	0.15	0.12	0.11
	corrupted, held-out total	113.6	114.4	114.9	107.5	106.7	106.0	117.0	111.1
ClassEval, Llama-3.1-8B, $3n$	selection tests	1.00	0.99	1.00	1.11	1.12	1.13	0.97	1.06
	held-out tests	1.00	0.99	1.00	1.07	1.09	1.09	0.99	1.04
	library pass (held-out)	0.14	0.13	0.13	0.19	0.20	0.20	0.17	0.16
	corrupted, held-out total	104.8	105.9	105.3	97.6	96.4	95.9	106.2	101.1

C.4 END-TO-END TREES: FULL RESULTS

Table 10: End-to-end trees on Qwen2.5-7B: corrupted-node fraction, 95% bootstrap interval over runs, and votes used per node. Chain-of-thought: 20 runs per tree and policy; direct: 40.

Policy	CoT, $B = 3n$	CoT, $B = 5n$	direct, $B = 3n$	direct, $B = 5n$
uniform	0.120 [0.09,0.16] (3.0)	0.073 [0.05,0.10] (5.0)	0.197 [0.17,0.23] (3.0)	0.200 [0.17,0.23] (5.0)
weight-proportional	0.133 [0.10,0.16] (3.0)	0.070 [0.05,0.10] (5.0)	0.190 [0.16,0.22] (3.0)	0.180 [0.15,0.21] (5.0)
structure-only (nominal rate)	0.096 [0.07,0.12] (3.0)	0.051 [0.03,0.07] (5.0)	0.205 [0.18,0.24] (3.0)	0.193 [0.16,0.22] (5.0)
water-filling (oracle curves)	0.108 [0.08,0.14] (3.0)	0.060 [0.04,0.09] (5.0)	0.209 [0.18,0.24] (3.0)	0.207 [0.18,0.24] (5.0)
adaptive, uniform caps	0.096 [0.07,0.13] (2.4)	0.072 [0.04,0.10] (2.5)	0.189 [0.16,0.22] (2.2)	0.204 [0.18,0.23] (2.2)
adaptive, structure-only caps	0.082 [0.05,0.12] (2.5)	0.066 [0.04,0.10] (2.5)	0.208 [0.18,0.24] (2.2)	0.198 [0.17,0.23] (2.2)
adaptive, oracle caps	0.054 [0.03,0.08] (2.4)	0.070 [0.04,0.10] (2.5)	0.210 [0.18,0.24] (2.1)	0.213 [0.19,0.24] (2.2)

C.5 TOKEN COSTS AND ANSWERING MODES

On Qwen2.5-7B, 30 mixed-difficulty multiplication problems were answered in two modes: a direct answer costs 4.7 tokens per vote on average, and a chain-of-thought answer 245 ($52\times$), with single-vote accuracy 0.48 and 0.77. At a matched token budget, a cost-aware allocation that chooses each node's mode and vote count greedily along the lower convex hull of its (tokens, error) points, restricted to odd vote counts, is compared with the better of all-direct and all-chain-of-thought. This greedy is an implementable integer allocation, not the continuous optimum of Proposition 9. The mode curves were estimated from the same samples used for scoring, so this study is in-sample.

Table 11: Cost-aware mode mixing: gain over the better single mode at matched tokens.

Token budget per node	gain	95% CI	nodes on CoT
$10\times$ direct	$1.29\times$	[1.27, 1.32]	13%
$20\times$ direct	$1.47\times$	[1.43, 1.51]	28%
$40\times$ direct	$1.57\times$	[1.53, 1.62]	48%
$80\times$ direct	$1.54\times$	[1.49, 1.60]	54%
$160\times$ direct	$1.43\times$	[1.38, 1.49]	54%
$320\times$ direct	$1.03\times$	[1.02, 1.04]	53%

C.6 CODE WITH A REALISTIC VERIFIER: FULL RESULTS

Table 12 gives both endpoints for every policy at $2\times$ and $3\times$ the initialization tokens, Table 13 the paired contrasts, and Table 14 the sensitivity analyses. Figure 4 shows whole-task success against tokens; every curve starts at the shared initial state.

Table 12: Whole-task success / fraction of components passing final tests. "init" is the shared initial state (tokens $= 1$). Columns are policies; rows give the budget in multiples of the initialization tokens.

Setting	budget	init	unif.	weight	struct.	blind	reach	degree	shuffled
Pilot suite, Qwen	$2\times$	0.39/0.72	0.46/0.79	0.46/0.79	0.46/0.79	0.52/0.82	0.53/0.84	0.53/0.84	0.49/0.80
	$3\times$		0.51/0.82	0.49/0.82	0.50/0.82	0.56/0.85	0.56/0.85	0.56/0.85	0.55/0.84
Pilot suite, Llama	$2\times$	0.14/0.53	0.22/0.64	0.22/0.64	0.22/0.64	0.38/0.72	0.39/0.73	0.38/0.72	0.35/0.68
	$3\times$		0.32/0.70	0.29/0.69	0.31/0.70	0.44/0.76	0.47/0.78	0.47/0.78	0.42/0.74
Pilot ClassEval, Qwen	$2\times$	0.22/0.60	0.25/0.65	0.25/0.65	0.25/0.65	0.32/0.67	0.32/0.67	0.32/0.67	0.31/0.67
	$3\times$		0.28/0.68	0.28/0.68	0.29/0.68	0.36/0.71	0.35/0.70	0.35/0.70	0.35/0.70
Pilot ClassEval, Llama	$2\times$	0.10/0.45	0.12/0.53	0.12/0.53	0.12/0.53	0.18/0.54	0.17/0.55	0.17/0.56	0.17/0.52
	$3\times$		0.16/0.57	0.16/0.57	0.16/0.57	0.22/0.59	0.23/0.60	0.23/0.60	0.21/0.57
Validation, Qwen	$2\times$	0.20/0.50	0.26/0.56	0.26/0.56	0.26/0.56	0.33/0.59	0.35/0.61	0.33/0.60	0.31/0.57
	$3\times$		0.29/0.58	0.30/0.60	0.29/0.59	0.37/0.63	0.38/0.64	0.37/0.64	0.34/0.61
Validation, Llama	$2\times$	0.13/0.47	0.20/0.56	0.20/0.56	0.20/0.56	0.28/0.64	0.36/0.72	0.36/0.72	0.27/0.59
	$3\times$		0.27/0.65	0.26/0.66	0.26/0.65	0.43/0.73	0.50/0.80	0.50/0.79	0.39/0.67

Table 13: Paired contrasts at $3\times$ the initialization tokens, in percentage points, with task-level bootstrap 95% intervals. Upper line of each setting: whole-task success; lower line: component fraction.

Setting	endpoint	reach − blind	degree − blind	reach − degree	reach − shuffled	blind − uniform	struct. − uniform
Pilot suite, Qwen	task	+0.0 [+0.0,+0.0]	+0.0 [+0.0,+0.0]	+0.0 [+0.0,+0.0]	+1.2 [+0.0,+3.1]	+5.6 [+0.6,+13.8]	-0.6 [-1.9,+0.0]
	fraction	+0.0 [+0.0,+0.0]	+0.0 [+0.0,+0.0]	+0.0 [+0.0,+0.0]	+1.6 [+0.4,+3.2]	+3.0 [+1.3,+5.0]	-0.3 [-0.9,+0.0]
Pilot suite, Llama	task	+3.1 [-0.6,+7.5]	+3.1 [-0.6,+7.5]	+0.0 [+0.0,+0.0]	+5.0 [+1.2,+9.4]	+12.5 [+6.2,+19.4]	-0.6 [-1.9,+0.0]
	fraction	+1.9 [-0.1,+4.3]	+2.1 [+0.2,+4.4]	-0.3 [-0.8,+0.0]	+3.9 [+1.7,+6.5]	+6.5 [+4.1,+9.0]	-0.1 [-0.3,+0.0]
Pilot ClassEval, Qwen	task	-0.8 [-1.7,+0.0]	-0.8 [-1.7,+0.0]	+0.0 [+0.0,+0.0]	-0.4 [-0.9,+0.0]	+7.5 [+3.2,+12.6]	+0.4 [+0.0,+1.1]
	fraction	-1.3 [-2.7,-0.0]	-1.4 [-2.8,-0.1]	+0.1 [+0.0,+0.3]	-0.2 [-1.3,+0.8]	+2.9 [+1.6,+4.5]	+0.0 [-0.2,+0.2]
Pilot ClassEval, Llama	task	+1.1 [+0.0,+2.3]	+1.1 [+0.2,+2.3]	+0.0 [+0.0,+0.0]	+2.3 [+0.8,+4.5]	+5.3 [+1.9,+9.6]	-0.2 [-0.6,+0.0]
	fraction	+1.0 [-0.5,+2.7]	+1.1 [-0.2,+2.5]	-0.1 [-0.5,+0.3]	+3.4 [+1.8,+5.6]	+2.0 [+1.0,+3.2]	-0.0 [-0.3,+0.4]
Validation, Qwen	task	+1.0 [+0.0,+3.0]	+0.0 [+0.0,+0.0]	+1.0 [+0.0,+3.0]	+4.0 [+0.0,+10.0]	+8.0 [+0.0,+19.0]	+0.0 [+0.0,+0.0]
	fraction	+1.3 [+0.0,+3.3]	+0.7 [-1.0,+2.9]	+0.6 [+0.0,+1.4]	+3.7 [+0.3,+9.7]	+5.3 [+1.3,+9.3]	+1.3 [+0.0,+3.9]
Validation, Llama	task	+7.0 [+0.0,+17.0]	+7.0 [+0.0,+19.0]	+0.0 [+0.0,+0.0]	+11.0 [+0.0,+25.0]	+16.0 [+6.0,+29.0]	-1.0 [-3.0,+0.0]
	fraction	+7.1 [+0.7,+16.0]	+6.7 [+0.4,+16.4]	+0.4 [+0.0,+1.3]	+13.1 [+4.1,+26.1]	+8.0 [+3.3,+13.6]	-0.1 [-0.4,+0.0]

Table 14: Sensitivity of the main code quantities to the cost model (pilot) and to the test-run limit. Token ratios are to reach uniform's whole-task success at 3×; a resample in which the initial state already meets the target counts as a ratio of 1 ("n.r.": not reached). The last column counts development/final runs killed by the limit.

Setting	variant	reach−blind (task, pp)	reach/uniform tokens	reach/blind tokens	killed runs
Pilot suite, Qwen	primary	+0.0 [+0.0,+0.0]	0.42 [0.39,0.48]	0.96 [0.86,1.00]	21/10
Pilot suite, Qwen	+80 overhead	+0.0 [+0.0,+0.0]	0.42 [0.39,0.48]	0.96 [0.86,1.00]	23/10
Pilot suite, Qwen	flat cost	+0.0 [+0.0,+0.0]	0.41 [0.38,0.46]	0.93 [0.83,1.00]	21/10
Pilot suite, Qwen	5 s wall-clock limit	+0.0 [+0.0,+0.0]	0.42 [0.39,0.48]	0.96 [0.86,1.00]	21/10
Pilot suite, Llama	primary	+3.1 [-0.6,+7.5]	0.47 [0.43,0.51]	0.99 [0.98,1.00]	69/27
Pilot suite, Llama	+80 overhead	+3.1 [-0.6,+7.5]	0.47 [0.43,0.51]	0.99 [0.98,1.00]	69/27
Pilot suite, Llama	flat cost	+1.9 [-0.6,+4.4]	0.48 [0.44,0.52]	0.96 [0.90,0.99]	67/27
Pilot ClassEval, Qwen	primary	-0.8 [-1.7,+0.0]	0.49 [0.46,0.54]	1.02 [1.00,1.06]	516/22
Pilot ClassEval, Qwen	+80 overhead	-0.8 [-1.7,+0.0]	0.49 [0.46,0.54]	1.02 [1.00,1.06]	503/21
Pilot ClassEval, Qwen	flat cost	-0.8 [-1.7,+0.0]	0.50 [0.46,0.57]	1.02 [1.00,1.06]	515/21
Pilot ClassEval, Llama	primary	+1.1 [+0.0,+2.3]	0.59 [0.52,0.71]	1.03 [0.92,1.15]	84/39
Pilot ClassEval, Llama	+80 overhead	+1.1 [+0.0,+2.3]	0.59 [0.52,0.71]	1.03 [0.92,1.15]	84/47
Pilot ClassEval, Llama	flat cost	+1.1 [+0.0,+2.3]	0.57 [0.49,0.69]	1.08 [0.95,1.25]	84/43
Validation, Qwen	primary	+1.0 [+0.0,+3.0]	0.46 [0.42,1.00]	1.00 [0.99,1.01]	0/0
Validation, Qwen	5 s wall-clock limit	+1.0 [+0.0,+3.0]	0.46 [0.42,0.71]	1.00 [0.99,1.01]	49/6
Validation, Llama	primary	+7.0 [+0.0,+17.0]	0.45 [0.41,0.51]	1.00 [0.76,1.01]	21/0

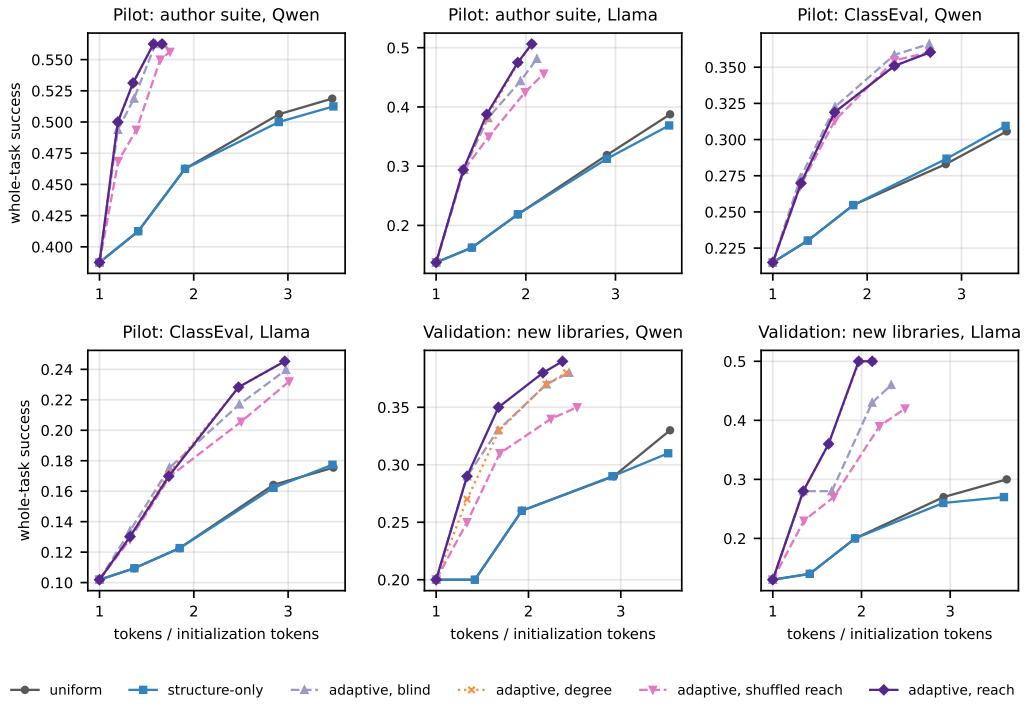

Figure 4: Whole-task success against tokens spent, relative to the initialization tokens, under the realistic verifier. Points are the initial state and the checkpoints at 1.5, 2, 3, and 4 times the initialization tokens. Adaptive policies stop once development tests pass, so their curves end to the left.

C.7 COMPARING REACH AND DEGREE ON CODE (EXPLORATORY)

These analyses were run after all primary results and change no endpoint. Table 15 lists, for each validation library, the graph-only criterion of Corollary 7, the exact integer comparison (water-filling on degree against water-filling on reach, both scored on the reach surrogate, $1 \le b_v \le 10$, nominal rate $\ln 2$, budget $3n$), the share of steps in which reach and degree would choose different components from the same state (replaying both trajectories with development feedback only), and the paired reach−blind whole-task effect at $3\times$. The whole-task effects are concentrated: Llama's mean comes from two libraries and Qwen's from one. On the pilot tasks the criterion has median 1.011 (author suite) and 1.000 (ClassEval). This criterion measures a continuous homogeneous-surrogate penalty, not an adaptive performance bound. The graph families we inspected did not justify another small constructed benchmark; we claim neither an extremal bound over graphs nor a statistical power limit.

Table 15: Reach versus degree on the validation libraries. Criterion: $\mathrm{AM}(w/d)/\mathrm{GM}(w/d)$. Integer: surrogate penalty of degree-based integer water-filling at budget $3n$. Differ: share of same-state steps in which reach and degree choose different components. Last columns: reach−blind whole-task effect (pp) at $3\times$.

Library	criterion	integer	differ (Llama)	differ (Qwen)	Δ task (Llama)	Δ task (Qwen)
textstats	1.125	1.000	50%	61%	+0	+0
linalg	1.062	1.000	6%	19%	+0	+0
rational	1.066	1.029	0%	35%	+0	+0
graphdist	1.136	1.000	40%	47%	+0	+0
zstrings	1.055	1.059	29%	22%	+0	+0
orderstats	1.127	1.018	0%	7%	+0	+0
polyint	1.011	1.000	19%	12%	+50	+0
bits	1.011	1.000	0%	0%	+0	+0
caldays	1.146	1.071	0%	–	+20	+0
inventory	1.174	1.033	29%	25%	+0	+10

Budget grid (exploratory). Because many adaptive runs stop drawing before $3\times$ (Appendix C.7), we reran every policy on the same candidate streams with a fixed grid of checkpoints (1.1, 1.25, 1.5, 1.75, 2, 2.5, 3, and 4 times the initialization tokens; trajectories are identical, only the saved states differ). Figure 5 shows the whole curve of paired differences. Differences are zero at $1.1\times$, where there is almost no budget beyond initialization; they appear from $1.5\times$; and on settings where runs stop drawing early they shrink again at larger budgets (for example the author suite with Qwen, where the difference returns to zero by $3\times$). Reach and degree give similar results throughout. This analysis was run after the primary results and is exploratory.

C.8 INDEPENDENT PUBLIC TASKS: REPOCLASSBENCH (EXPLORATORY)

RepoClassBench (Deshpande et al., 2024) asks for a complete class inside a real repository. Its Python environments are built per repository; we built only the litestar repository (41 of the 97 Python tasks, one commit) in a Python 3.9 virtual environment. One class (WebSocket) served as a feasibility probe and is excluded. Tasks were chosen by a rule fixed before any candidate for them was generated: litestar classes with 5–14 methods (overload groups collapsed), at least two intra-class call edges, and at least six test directives; a seeded shuffle; kept in that order if at least six tests pass with the reference and at least three methods are exercised by a passing test; stopping at six. Seven classes passed the static filter and one failed the reference check. Components are the class's methods, and the graph is the reference class's intra-class calls, analogous to ClassEval's declared dependencies. Tests are assigned to methods by tracing, under the reference class, which methods each test calls; this assignment is fixed before generation and no reference code is shown to any policy. Reference-passing tests are split once into development and final halves. Development feedback runs the development tests on the policy's current class, with every method replaced by its current generated candidate, in a private copy of the repository. Candidates (8 per method, temperature 0.8, at most 700 tokens) see the class description, the module imports, and the class with all method bodies hidden; costs are exact token counts. Every test has a $10\,\mathrm{s}$ limit. All other rules, the policies, and the budget grid are as in the main code study, with 5 repetitions per task. The selected graphs

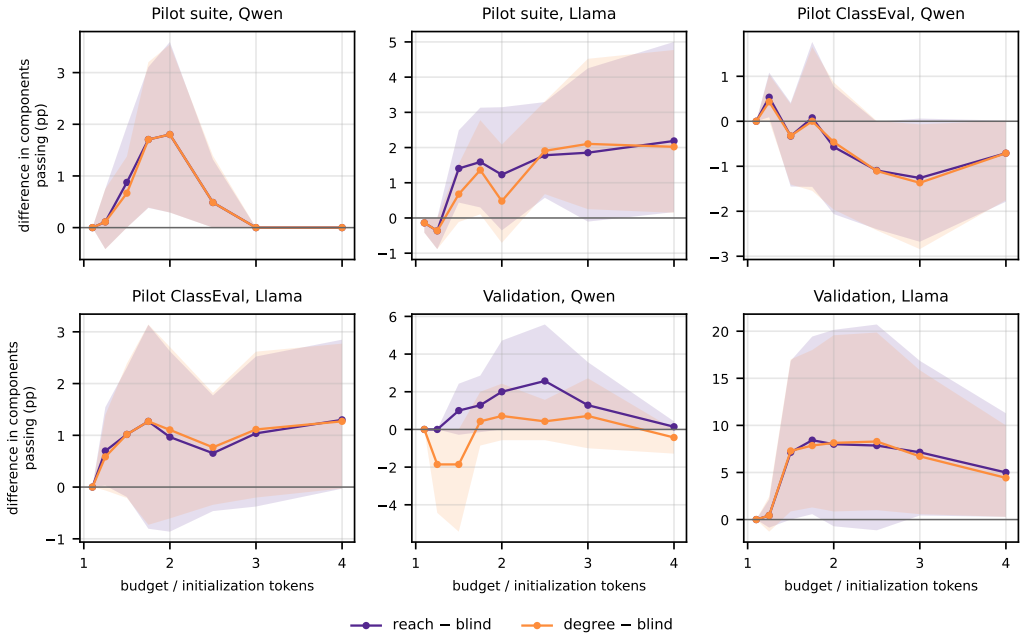

Figure 5: Paired differences in the fraction of components passing final tests, adaptive-reach minus adaptive-blind and adaptive-degree minus adaptive-blind, across a fixed grid of budgets (task-level bootstrap 95% bands).

are nearly flat (Table 16), so the surrogate predicts small structural gains and almost no difference between reach and degree.

Table 16: Selected RepoClassBench classes (litestar).

Class	methods	max reach	AM(w)/GM(w)	AM(w/d)/GM(w/d)	dev. tests	final tests
ClientSideSessionBackend	7	3	1.09	1.000	3	3
ServerSideSessionBackend	7	2	1.06	1.000	9	9
SchemaRegistry	8	2	1.05	1.000	5	5
AbstractDTO	14	4	1.14	1.006	14	14
HTTPRouteHandler	14	3	1.12	1.015	78	77
UploadFile	7	5	1.25	1.000	10	10

C.9 Idealized repair diagnostic

For each task, we use the initial assemblies of repetitions 0–4, with the allocation runs' candidate streams and seeds. We replace each component in turn by its reference implementation, hold the others fixed, and rerun final tests. A repair's value is the net increase in passing components, subtracting regressions. Direct and downstream fixes count fail-to-pass changes in the repaired component and other components, respectively.

Predictors use reach, degree, the local development-test failure signal, or their products; a final-test failure signal is an oracle reference. For every predictor, ties are scored by the mean gain over all tied top choices. Conditional top-1 efficiency divides this gain by the best single repair's gain in states where that best gain is positive. We average within tasks and bootstrap tasks. The unconditional score includes all states, including those where no repair helps; it measures net additional passing components. Tables report both scores.

Candidate coverage uses every tied best target in each helped state, counting repeated occurrences across states. We test its candidates with reference dependencies on final tests. The mean pass rate

includes zero-coverage targets; it is not conditional on a passing candidate existing. This analysis makes no model calls and uses reference code to measure idealized repair value, not deployable savings.

Table 17: Repair diagnostic, full results. Top-1 efficiency with task-bootstrap 95% intervals (RepoClassBench scores omitted: at most three helped states); last two columns: paired differences of reach×local against local and degree×local.

Setting	helped	downstream	regressions	local	degree×local	reach×local	reach−local	reach−degree
ClassEval, Llama	228/265	25%	22	0.56 [0.49,0.64]	0.69 [0.59,0.79]	0.71 [0.61,0.80]	+0.15 [+0.07,+0.23]	+0.02 [+0.00,+0.04]
ClassEval, Qwen	197/265	21%	44	0.68 [0.60,0.76]	0.74 [0.63,0.83]	0.75 [0.64,0.84]	+0.07 [-0.01,+0.14]	+0.01 [+0.00,+0.03]
Validation, Llama	42/50	40%	3	0.43 [0.29,0.60]	0.81 [0.64,0.94]	0.86 [0.79,0.93]	+0.42 [+0.26,+0.57]	+0.05 [-0.06,+0.19]
Validation, Qwen	40/50	59%	8	0.49 [0.34,0.67]	0.61 [0.38,0.81]	0.83 [0.60,1.00]	+0.34 [+0.13,+0.55]	+0.23 [-0.03,+0.48]
Author suite, Llama	71/80	30%	4	0.58 [0.47,0.70]	0.83 [0.74,0.92]	0.88 [0.80,0.94]	+0.29 [+0.20,+0.40]	+0.04 [+0.00,+0.09]
Author suite, Qwen	51/80	29%	0	0.57 [0.44,0.70]	0.79 [0.67,0.91]	0.85 [0.73,0.95]	+0.28 [+0.15,+0.41]	+0.05 [+0.00,+0.13]
RepoClassBench, Llama	3/30	78%	0	–	–	–	–	–
RepoClassBench, Qwen	1/30	100%	0	–	–	–	–	–

Table 18: Repair diagnostic over all states and link to candidate pools. Mean net increase in passing components per state (including states where no repair helps; task-bootstrap intervals), and, for all tied best-target occurrences in helped states, the fraction with no passing candidate and the mean fraction of passing candidates (including zero-coverage targets).

Setting	zero-benefit	best	local	degree×local	reach×local	no correct candidate	correct share
ClassEval, Llama	37/265	1.26	0.59 [0.51,0.67]	0.80 [0.63,0.99]	0.83 [0.65,1.01]	43%	0.22
ClassEval, Qwen	68/265	0.98	0.58 [0.49,0.67]	0.65 [0.51,0.77]	0.66 [0.54,0.79]	58%	0.20
Validation, Llama	8/50	1.74	0.58 [0.39,0.75]	1.43 [0.91,2.11]	1.43 [0.88,2.10]	12%	0.39
Validation, Qwen	10/50	2.00	0.76 [0.50,1.02]	1.13 [0.41,2.10]	1.82 [0.88,2.86]	57%	0.21
Author suite, Llama	9/80	1.48	0.70 [0.58,0.81]	1.23 [0.99,1.53]	1.30 [1.03,1.59]	10%	0.33
Author suite, Qwen	29/80	1.04	0.59 [0.37,0.85]	0.89 [0.60,1.19]	0.93 [0.62,1.24]	24%	0.34

C.10 SHARED-FACT AUDITING ON TAT-QA: FEASIBILITY CHECK (STOPPED)

We tested shared-fact auditing on 20 randomly selected TAT-QA (Zhu et al., 2021) development documents with at least two arithmetic questions. Qwen2.5-7B generated facts and expressions with greedy decoding. The prespecified criteria required fact reuse and beneficial source-grounded rechecks. Facts were reused in 14 documents. Of 103 checks, 19 changed values: none corrected an answer and 3 made a correct answer wrong (all checks: $31/55$ to $28/55$ correct). In 12 of 13 parsed wrong answers, fact values appeared verbatim in the source, but the intended year, entity, unit, and operation were unchecked; we cannot attribute the errors to arithmetic. We stopped the allocation pilot because checks did not help. Gold derivations supported descriptive analysis only, never a policy.