Adaptive Graph Harness Optimization for LLM Agents: Where Do the Gains Come From?

Yiwen Lu
University of Notre Dame
ylu37@nd.edu

Code and data: https://github.com/EvenEureka/harness-graph

Automatic harness optimization treats the control flow around an LLM agent as a graph and searches or adapts it to improve performance. However, reported gains are hard to attribute, since hindsight selection, run-to-run variance and extra compute can all look like structural improvement. In this paper, we evaluate three increasingly adaptive forms of harness-graph optimization, namely static graph search, pre-run per-task routing and mid-run trajectory switching, under one out-of-sample rule: every graph choice is scored on tasks, repositories or runs that did not inform it. Across 24,845 SWE-smith trajectories and 7,506 AppWorld runs, single-model graph adaptation shows no detectable gain in repair or task success over simple fixed baselines. A graph found by search in a space of 3,600 graphs is 0.2 points above a hand-written check-and-retry loop, and a frozen mid-run controller repairs 84.0% of tasks on unseen repositories against 85.6% for always continuing the conversation. In contrast, a pass/fail test check adds 14.8 points over unconditional retries, failure diagnostics add another 7.3, and continuing the same runs beyond call 50 under a 100-call cap adds 8.5. These results point harness engineering toward executed feedback, budget, model choice and stopping rules.

1 Introduction

An LLM agent consists of a foundation model and a *harness*: the surrounding control program that governs how the model is prompted, when its work is verified, and how execution proceeds after a failure. Recent work frames this harness as an executable graph and optimizes it automatically, by searching for a single workflow per benchmark (Zhuge et al., 2024; Hu et al., 2024; Zhang et al., 2024a; Lee et al., 2026), routing each task to a specialized harness (Li et al., 2026), or intervening mid-trajectory to restart runs that appear stuck (Wang et al., 2026). Each step makes the harness more adaptive, and each reports gains. This raises a basic question: when a graph optimizer reports a gain, how much of it comes from the graph?

The question is hard because of a pattern that pools of graphs show. As illustrated in Figure 1, choosing the best of three graphs for each task in hindsight looks 2.4 to 5.7 points better than a fixed graph, which makes per-task and mid-run adaptation look promising. Yet agent trajectories are stochastic: rerunning the same agent on the same issue already solves substantially more tasks in union than a single run (Zhang et al., 2024b), and in our SWE-smith data two runs of one graph disagree more than half as often as two different graphs. When the per-task choice is scored on a fresh run of the same tasks, the advantage shrinks to within one point. Consequently, the core question shifts from whether a pool of graphs contains a better choice in hindsight to whether a selector keeps that choice on a run it has not seen.

Current evaluations leave this attribution open for a second reason. A harness graph bundles several things: executed feedback such as a test run, the budget the agent is allowed to spend, the model that does the work, and the topology that connects them. Adaptive procedures routinely add retries, extra verification calls or compute along with a new topology. Without strict separation between the data used to choose a graph and the runs used to score it, and without a strong fixed baseline, an apparent topological gain can reflect selection bias, run-to-run variance or hidden compute.

To address this, we shift the focus from finding the best graph to measuring what is left of each gain under one evaluation rule. Our out-of-sample comparisons score a graph selected by search, routing or mid-run control on tasks, repositories or fresh runs that never informed the choice, against simple fixed baselines, with call caps, model calls and tokens reported.

Based on this rule, we study three increasingly adaptive regimes on SWE-smith bug repair and AppWorld app-API tasks. Static graph search explores a space of 3,600 workflow graphs to select one harness for all tasks. Pre-run routing freezes a feature-based router that picks a graph per task before execution. Mid-run trajectory switching, the most

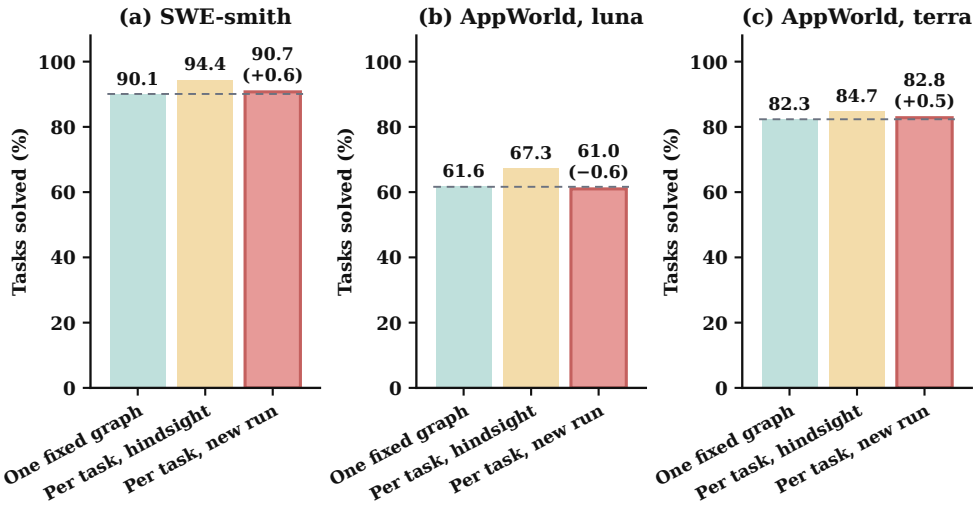


Figure 1: Per-task advantages seen in hindsight shrink to within one point on a fresh run. Three graphs per panel. Middle bar: the best graph per task, chosen and scored on the same runs. Right bar: chosen on earlier runs and scored on a new run of the same task, with its difference from the fixed graph in parentheses. Dashed line: one fixed graph.

adaptive regime, checkpoints open runs at call 20 and routes the remaining budget to one of four subgraphs: continuing the conversation, resetting it with a programmatic digest, forking into two branches, or stopping early. We evaluate this controller under a prospective protocol with seven held-out test repositories, pre-registered endpoints and four independently audited protocol gates.

Across 24,845 SWE-smith trajectories and 7,506 AppWorld runs, single-model graph adaptation shows no detectable gain in repair or task success over fixed baselines, while four other design choices produce large measured gains (Figure 2). The graph selected by search is 0.2 points above a two-node check-and-retry loop on 900 fresh tasks, and the frozen mid-run controller trails always continuing the same conversation by 1.6 points on unseen repositories. In contrast, a pass/fail test check adds 14.8 points over unconditional retries, failure diagnostics add another 7.3, and continuing the same runs from 50 to 100 calls adds 8.5. Overall, for agents with executable checks, these results point harness engineering toward how an executed check controls continuation, how long the agent continues in context, which model executes the task, and when a solved run is terminated.

2 Related Work

Searched and Per-Task Harness Graphs. ReAct interleaves reasoning with actions (Yao et al., 2022), and hand-built coding harnesses add structure around that loop (Yang et al., 2024; Xia et al., 2024). Automatic design made the structure the object of optimization: GPTSwarm optimizes prompts and graph connectivity (Zhuge et al., 2024), ADAS has a meta agent program new agents (Hu et al., 2024), AFlow searches code-represented workflows (Zhang et al., 2024a), and Meta-Harness searches coding-agent harnesses end to end (Lee et al., 2026). The next step makes the choice per task. After a base run, OO-Harness selects a specialized harness or reuses the base result; relative to the best global harness it gains 13.2 mean-reward points on AutomationBench, 1.1 on Workspace-Bench and 1.2 pass-rate points on Toolathlon-GYM (Li et al., 2026). A central evaluation pitfall in this literature is the oracle union: on SWE-bench Lite, ten different agents jointly solve 54.3% of issues against a 26.6% average, and ten reruns of a single agent jointly solve 34.7% to 38.0% against 15.9% to 21.7% (Zhang et al., 2024b). A survey of graph engineering accordingly asks that system-level improvements be separated from gains caused by retries or compute (Feng et al., 2026; Kapoor et al., 2024). We make out-of-sample comparison the evaluation rule and score every graph choice on runs or tasks excluded from making it, against a rerun of a fixed baseline.

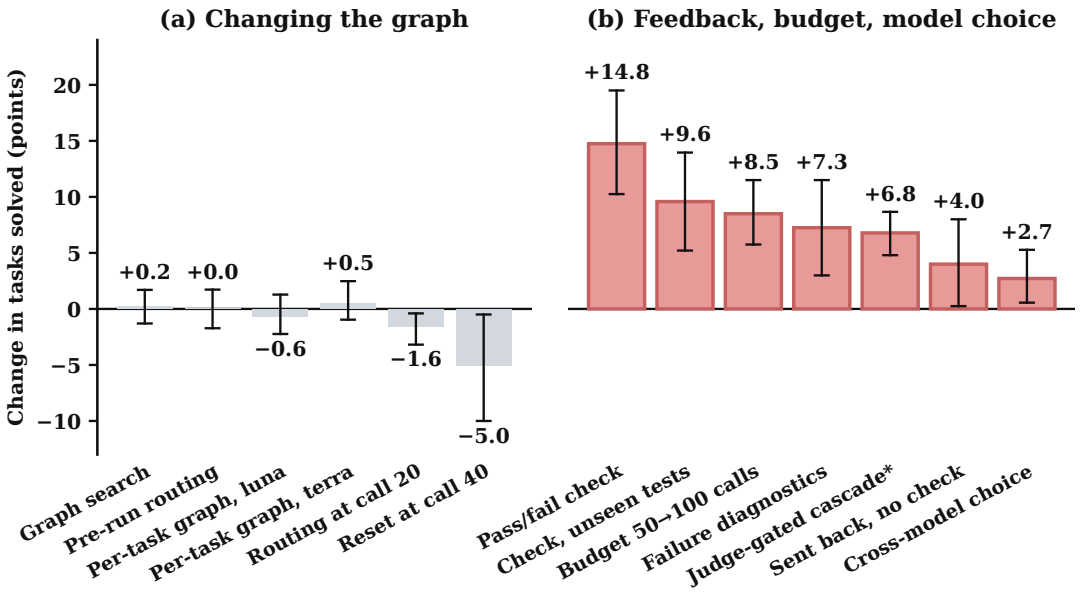


Figure 2: Where harness gains come from. Each bar is one paired comparison with its 95% bootstrap interval; (a) graph adaptation against a fixed baseline, (b) feedback, budget and model selection. Bars come from different experiments and task pools (Table 1); only the three sequential ablation contrasts add up. The starred bar is a stored-run replay.

Mid-Run Restart and Stopping. The most adaptive form of graph control acts inside a run. Fail-Fast, Restart-Smart aborts a run on a failure monitor's alarm and restarts with the interrupted diff offered as an overlay; with Qwen3.6-27B on SWE-bench Verified, at a 25% target false-positive rate, it raises resolution from 66.6% to 71.8% at 43.8% more compute, and a cold restart reaches 66.8% (Wang et al., 2026). Across 147,002 traces, early-prefix predictors discriminate between attempts on the same task at near-chance level, and uninterrupted continuation reaches higher solve rates at lower mean token cost than the evaluated restart policies (Srinivasan et al., 2026). Both operate on stored or monitored runs of a fixed harness, whereas we execute every branch from a shared snapshot, freeze the controller before the test, and add a fork action and a late checkpoint.

Executed Feedback and Compute. Reflexion returns an evaluator's result and a self-reflection to the model (Shinn et al., 2023); a baseline that only reruns GPT-4 while the provided tests fail reaches 92.0% on HumanEval (Kapoor et al., 2024); CodeMonkeys scales iterations per trajectory and trajectories per problem separately (Ehrlich et al., 2025). Generated tests are a weaker signal: of the 213 reproduction tests Agentless keeps on SWE-bench Lite, 94 report the issue resolved after the ground-truth patch (Xia et al., 2024; Mündler et al., 2024). These papers establish that feedback and compute matter. Our study instead measures how much of a check-and-retry loop's gain each component carries, and whether the gain holds on tests the agent could not see.

3 Problem Formulation

3.1 Harness-Graph Formulation

A harness graph g consists of nodes and edges over a shared state. A *node* is one unit of execution: an agent conversation in which the model issues shell commands until it submits, or a program such as a test run. A node reads and writes the state s, which holds the conversation, the working copy of the repository and the latest feedback, and returns an outcome label. An *edge* is a control rule: node X returned outcome r, so node Y runs next. A run of g on a task under a cap of B model calls yields a patch that is graded as repaired, $y = 1$, or not, together with its cost in calls and tokens.

Two graphs anchor the study. The *plain loop* is the single node `solve` with the edge `done` $\rightarrow$ submit. *Check-and-retry* adds the program node `check`, which runs the task's tests on the current patch. Its edges are: `solve` returns

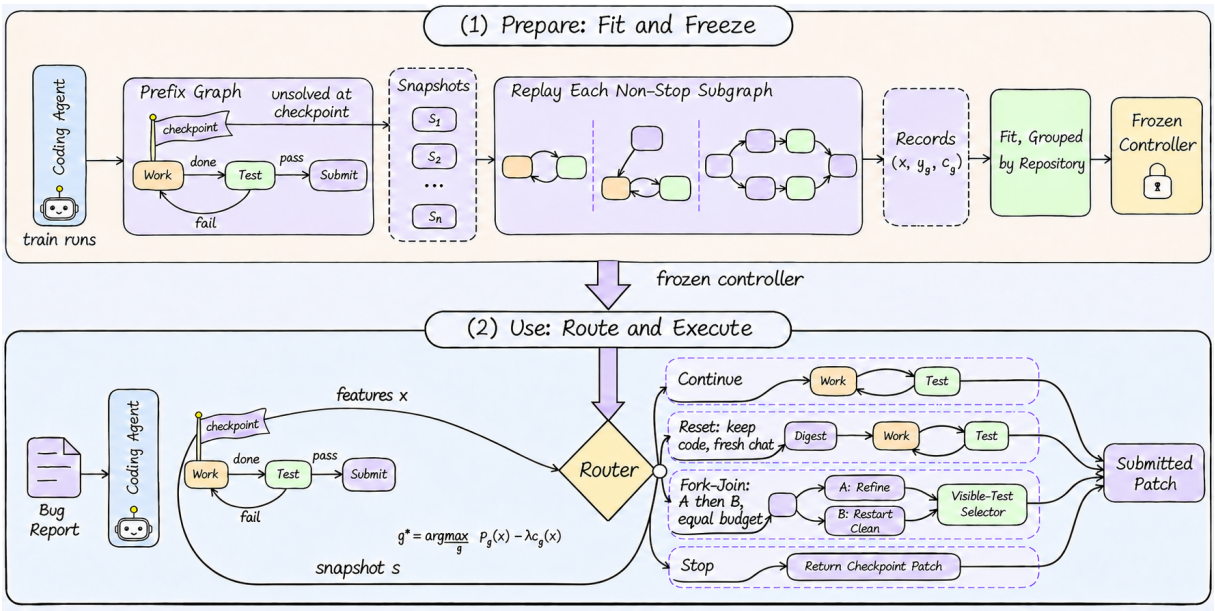


Figure 3: Mid-run trajectory switching. *Work* is the agent conversation, *Test* the program that runs the visible tests, and *Digest* a program that summarizes the repository state; edges are labelled with the outcome that triggers them. (1) Training runs execute check-and-retry to the checkpoint, and each non-stop subgraph is replayed from the snapshot to give a repair label y_g and a call cost c_g next to the trajectory features x. (2) At test time the frozen router picks $g^* = \arg\max_g P_g(x) - \lambda c_g(x)$.

`done` $\rightarrow$ `check`; `check` returns `pass` $\rightarrow$ submit, or `fail` $\rightarrow$ `solve` with the failure output written into the conversation, at most twice. The conversation is kept across retries.

3.2 Three Regimes of Graph Adaptation

Static Graph Search. Nine design variables, such as whether to plan first, whether the check is enforced and what follows a failed check, compile into 3,600 distinct graphs of 1 to 11 nodes. The space is explored in two ways with repair rate as the objective: a table that scores the two anchor graphs and 30 sampled graphs on 188 tasks, and ten live local searches of 512 agent runs each that climb from the plain loop one design variable at a time. The returned graphs and the anchors form 14 candidates, which are scored on 56 selection tasks, and the one with the highest repair rate goes to confirmation. Appendix B gives the procedures.

Pre-Run Routing. A router picks one of three graphs per task before the run: the plain loop, check-and-retry, or the graph selected by search. For each graph, a ridge regression predicts repair from five features of the task, namely test visibility, the log word count of the issue, and indicators for code formatting, for a traceback or error marker, and for a Python file name. The router runs the graph with the highest prediction. It is fitted on the 188 search tasks and frozen.

Mid-Run Trajectory Switching. The third regime decides after part of the trajectory is known (Figure 3). Every run follows check-and-retry up to call 20, and runs that pass the check earlier finish normally. For a run still open at that checkpoint, the harness stores the workspace and conversation and computes eight features x without a model call, such as whether a source file was edited, the latest visible pass ratio and the length of the current streak of identical errors. It then commits the remaining 34 calls to one of four subgraphs. CONTINUE keeps the conversation and the files. RESET-WITH-DIGEST keeps the files and starts a new conversation from a programmatic digest of the repository status and the latest test output. FORK-JOIN splits the calls between a branch that refines the existing patch and a branch that starts from the clean repository, and the visible tests pick the winner. EARLY-STOP submits the checkpoint patch. Logistic regressions predict each subgraph's repair probability $P_g(x)$ and ridge regressions its normalized extra calls

$c_g(x)$, and the controller chooses

$$g^*(x;\lambda) = \arg\max_g \; P_g(x) - \lambda\, c_g(x), \tag{1}$$

where $\lambda \geq 0$ is a cost penalty and stopping enters with $P = c = 0$. Two operating points are fixed by cross-validation grouped by repository: λ^* for accuracy and λ_{cons} for cost saving.

3.3 Out-of-Sample Evaluation Protocol

A choice is scored only on data that did not inform it: a graph selected by search is scored on held-out and fresh tasks, a per-task choice on a new run of the task, and a fitted controller on repositories it never saw. Differences are paired over tasks, and 95% intervals come from a bootstrap that redraws tasks inside repositories, whole repositories for the mid-run test, or whole scenarios on AppWorld.

The mid-run test is prospective. The 450 tasks are split by repository into 200 training tasks from 15 repositories and 250 test tasks from seven repositories on which no agent had run and nothing had been tuned. On the training tasks all three non-stop subgraphs run from the same snapshot, so each open run yields three outcomes that share a prefix. The controller and its configuration are hashed before any agent runs on a test repository, two endpoints are registered in advance, and four protocol gates covering repository separation, leakage controls, the frozen artifacts and the statistical implementation are audited independently with Codex. Appendix A gives the endpoints, the gate verdicts and the deviations from the protocol.

4 Experiments

4.1 Experimental Setup

Datasets. SWE-smith synthesizes bugs that break existing tests in real Python repositories (Yang et al., 2025). A run starts in a fresh container with the issue text and the buggy repository, and a patch counts as a repair when every designated test passes in a clean copy. The failing tests are visible to the agent, SWE-smith's default, unless stated. AppWorld poses tasks over nine simulated apps and grades the final database state (Trivedi et al., 2024); we use its 417 Test-Challenge tasks. Table 1 lists the experiments, which comprise 24,845 scored SWE-smith trajectories and 7,506 AppWorld runs.

Baselines. The fixed baselines are (1) the plain loop and check-and-retry for static search and pre-run routing, (2) always applying one subgraph, a uniform call cap and random routing with matched action shares for mid-run switching, and (3) on AppWorld, incremental execution, in which the model writes a piece of code, runs it and reads the output, compared with plan-then-execute and whole-program-then-rewrite graphs.

Implementation Details. All SWE-smith runs use Qwen3.8-27B served locally with a cap of 50 model calls unless stated; thinking is disabled because a preliminary run with thinking enabled showed no detectable repair gain and ran six to seven times slower. AppWorld uses two hosted models, gpt-6-luna and gpt-6.1-terra, with a cap of 50 calls.

Evaluation Metrics. We report the repair rate on SWE-smith and the task success rate on AppWorld, with mean model calls and tokens per task for the main comparisons, and dollar cost for the AppWorld cascade.

4.2 Static Search and Pre-Run Routing

On 900 fresh tasks from 15 repositories that no earlier step had used, the graph selected by search repairs 81.5% against 81.3% for check-and-retry, a difference of $+0.2$ points $[-1.3, 1.7]$. The selected graph is check-and-retry plus a `reflect` node after a failed check and a `compact` node entered when the agent stalls; it led by 1.8 points on the 56 selection tasks and is $+0.6$ points above check-and-retry $[-1.3, 2.8]$ on the 232 held-out tasks, while using 3.7% fewer tokens per run $[1.1, 6.2]$ on the fresh tasks. The check itself matters far more: check-and-retry repairs 90.7% of the

Experiment	Setting	Tasks	What is compared, and on what the choice is scored
Graph adaptation			
Search and selection	SWE-smith	188 + 56	32-graph table, ten live searches, 14 candidates; used only to choose
Confirmation	SWE-smith	232; 900	selected graph, check-and-retry, plain loop; tasks unseen by search
Pre-run routing	SWE-smith	232	frozen router against one fixed graph; held-out tasks
Graphs and models	AppWorld	417	three graphs $\times$ two models, three runs each; choice scored on a held-out run
Mid-run switching	SWE-smith	200 / 250	controller fitted on 15 repositories, frozen, tested on 7 unseen repositories
Late checkpoint	SWE-smith	200	continue against restart at call 40 under an 80-call cap; training tasks
Feedback and budget			
Withheld tests	SWE-smith	240 ($\times$2 runs)	check-and-retry against plain loop; scored on tests removed from the workspace
Ablation	SWE-smith	200 ($\times$2 runs)	plain loop, forced continuation, pass/fail gate, full gate
Budget extension	SWE-smith	200 ($\times$2 runs)	check-and-retry at 100 calls; patch after call 50 against the final patch

Table 1: Experiments. The shaded row is the prospective test of the most adaptive regime, with 200 training and 250 test tasks. The 900 fresh tasks each combine several defects and are harder than the 232 held-out tasks; the three feedback and budget experiments draw their tasks from the 900.

held-out tasks against 84.3% for the plain loop, $+6.5$ points [3.4, 9.7], and in a single run on the fresh tasks 80.4% against 56.2%, $+24.2$ points [20.9, 27.3]. All ten live searches end on a graph with the enforced check. The selected graph therefore adds no detectable repair gain over check-and-retry.

Pre-run routing gives the same picture. On the 232 held-out tasks, with the visible-test and hidden-test conditions weighted equally, the fixed graph selected on the training data repairs 77.7%, a rule on test visibility 77.8% and the feature router 77.7%, so the router minus the fixed graph is 0.0 points [-1.7, 1.7]. Picking the best graph per task in hindsight would give 82.2%. The router therefore shows no detectable gain over the fixed graph on held-out tasks.

4.3 Hindsight versus Fresh Runs

Figure 1 tests how much of a hindsight gap a selector keeps on a new run. With two runs of three graphs on the 232 held-out SWE-smith tasks, the per-task best in hindsight reaches 94.4% against 90.1% for the best fixed graph, while the graph picked per task on one run and scored on the other reaches 90.7%. AppWorld repeats the pattern within a single model. With three runs of each of three graphs per task, we choose the graph from two runs and score it on the third, holding each run out in turn (Appendix C gives the tie rule). The choice changes success by -0.6 points [-2.2, 1.3] for gpt-6-luna and $+0.5$ points [-1.0, 2.5] for gpt-6.1-terra over the fixed incremental graph, where the hindsight gaps are 5.7 and 2.4 points. On the SWE-smith tasks, two runs of the same graph already disagree on 5.2% of tasks, against 8.3% for two different graphs. In these three comparisons, the fresh-run point estimates are all within one point of their fixed baselines.

4.4 Mid-Run Trajectory Switching

Training-Set Results. Of the 200 training tasks, 194 are usable and 141 runs are open at call 20. Always continuing repairs 82.0% of the 194 tasks, always resetting 69.1%, always forking 61.9%, and submitting the call-20 patch 37.6%. CONTINUE fails on 35 open runs, and RESET-WITH-DIGEST or FORK-JOIN repairs 3 of them, so the best of three in hindsight reaches 83.5%. On these recorded runs, any rule that chooses among the three subgraphs at call 20 is therefore bounded at 1.5 points above always continuing, and the cross-validated controller sends 92% of open runs to CONTINUE.

Test-Set Results. Table 2 shows the prospective test. Always continuing has the highest repair rate, 85.6%, and the frozen controller at λ^* trails it by 1.6 points [-3.2, -0.4]. The controller routes 94.3% of open runs to CONTINUE, 2.3% to RESET-WITH-DIGEST and 3.4% to FORK-JOIN; relative to their own CONTINUE branches, these ten diverted

Policy at call 20	Repair	Calls	Tokens	Repair vs. always CONTINUE
Always CONTINUE	**85.6%**	31.0	353k	—
Frozen controller, λ^*	84.0%	31.3	345k	-1.6 $[-3.2, -0.4]$
Frozen controller, λ_{cons}	84.4%	31.2	345k	-1.2 $[-2.0, -0.4]$
Random routing, same shares	84.6%	31.4	—	-1.0
Always RESET-WITH-DIGEST	71.6%	34.9	286k	-14.0
Always FORK-JOIN	68.0%	39.1	266k	-17.6
Submit the call-20 patch	44.4%	17.9	115k	-41.2

Table 2: Prospective mid-run test on 250 tasks from seven unseen repositories. Calls and tokens are means per task over the whole run; differences are in points, with 95% repository-bootstrap intervals for the two controller rows. 176 runs are open at call 20.

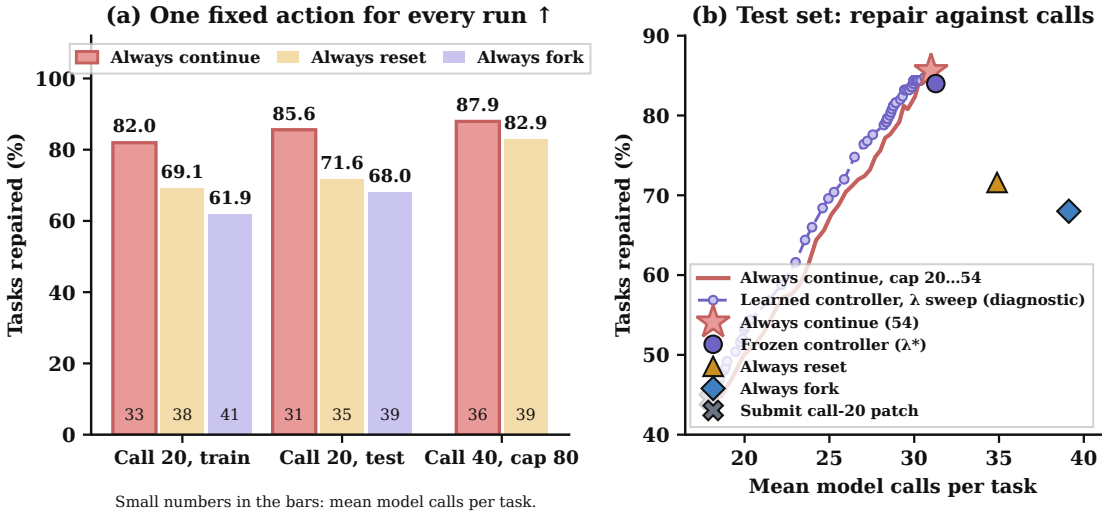


Figure 4: Mid-run switching. (a) Repair rate when every open run takes the same action, at call 20 on the training and test sets and at call 40 under an 80-call cap; the numbers inside the bars are mean calls. (b) Test set: always continuing truncated at every uniform cap, and a diagnostic sweep of the learned controller over the cost penalty λ.

runs yield four fewer repairs and 70 more calls. Both registered endpoints are missed. For Endpoint A, no uniform cap up to 54 calls reaches the controller's mean calls, so the reported unmatched cap-54 baseline is always continuing itself, and the controller trails random routing with the same shares by 0.6 points $[-1.3, 0.1]$; for Endpoint B, the controller at λ_{cons} is 1.2 points below always continuing $[-2.0, -0.4]$ and saves 2.4% of tokens $[-0.1, 5.3]$ while spending 0.5% more calls. The ordering of the fixed actions matches the training set (Figure 4a), and the hindsight bound is again small: of the 36 open runs on which CONTINUE fails, another subgraph repairs 5, so selecting among the recorded branches reaches 87.6%. These results show that, on the recorded test runs, continuing has the highest fixed-action repair rate, and hindsight selection among the three non-stop branches gains at most 2.0 points over always continuing.

4.5 Ablation Study

Effect of Executed Feedback. Full check-and-retry repairs 79.3% of 200 tasks against 53.3% for the plain loop (Figure 5a). Four variants that share the solver prompt, the continuation prompt, the call cap and the retry limit separate the steps in between. Forced continuation, which sends the agent back twice without running a check, reaches 57.3%, a gain of 4.0 points [0.3, 8.0]. A pass/fail gate, which runs the check and tells the agent only that it failed, reaches 72.0%, a further 14.8 points [10.3, 19.5]. Adding the failure output gives the full gate and another 7.3 points [3.0, 11.5], the comparison fixed in advance as primary. The two gated variants spend about as many calls as forced continuation, with means of 34.5 and 33.4 against 33.1. These results show that most of check-and-retry's lead lies beyond unconditional retries, in variants where the outcome of a check decides whether the agent is sent back and what it is told.

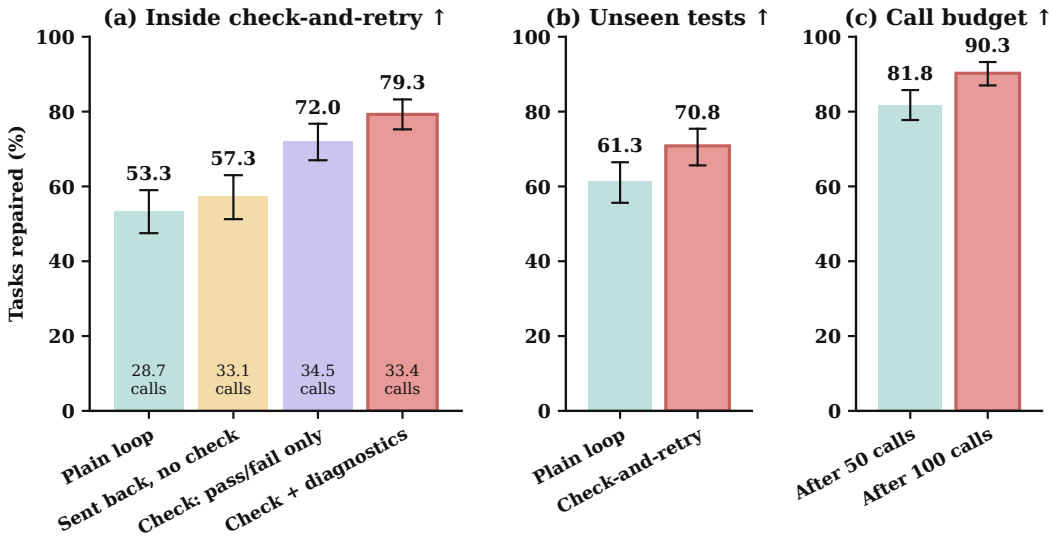


Figure 5: What check-and-retry contributes. (a) Four ablation variants on 200 tasks with two runs each; the text inside each bar is the mean number of model calls. (b) Patches that pass the withheld failing tests and every previously passing test, on 240 tasks. (c) The patch after call 50 against the final patch under a 100-call cap, on 200 tasks. Whiskers are 95% bootstrap intervals.

Transfer to Withheld Tests. Check-and-retry also leads on tests the agent never saw. For 240 tasks, failing-test files are grouped by import connections and about half of the groups are removed from the workspace, so that neither the agent nor the check node can run them; the shown tests still detect every component of the bug patch, defined by its edited functions. Scored on the withheld failing tests plus every previously passing test, check-and-retry reaches 70.8% against 61.3% for the plain loop, a difference of 9.6 points [5.2, 14.0] (Figure 5b). Among patches that pass the shown failing tests and the previously passing tests in the workspace, the withheld tests reject 22.3% for check-and-retry and 22.8% for the plain loop, and in a separate subset of 92 tasks whose shown tests do not detect every component the difference is $+1.6$ points $[-3.3, 6.5]$. The withheld-test lead is thus established in this component-detecting subset.

Effect of the Call Budget. Of 336 unsolved check-and-retry runs on the 900 fresh tasks, 302 end on the 50-call cap. In 400 new runs with a 100-call cap, the patch after call 50 is a repair in 81.8% of cases and the final patch in 90.3%, a gain of 8.5 points [5.8, 11.5] with 28 tasks better and none worse, for 49.6% more tokens (Figure 5c). Runs that continue in the same conversation thus keep producing repairs between call 50 and call 100.

4.6 Model Escalation and Early Stopping

Cross-Model Selection. In the evaluated AppWorld pool, per-task selection across models keeps a gain on fresh runs. The hold-one-run-out procedure of Section 4.3 applied to all six model and graph configurations reaches 85.1% against 82.3% for the best fixed configuration, a gain of 2.7 points [0.6, 5.3], whereas the two single-model estimates have intervals that include zero. With the incremental graph, gpt-6-luna succeeds on 61.6% of runs at $0.0028 per task and gpt-6.1-terra on 82.3% at $0.0233.

Judge-Gated Model Cascade. A cascade uses this cost difference. It runs luna first; a judge, luna again, reads the instruction and the transcript without the grade, and a verdict of NOT COMPLETED starts terra on the same task in a fresh world. Replayed on the 1,251 stored luna runs with no fitted parameter, the cascade succeeds on 80.4% of tasks at $0.0176 per task, judge calls included, against 82.3% at $0.0233 for terra alone. This is 1.9 points lower $[-3.9, -0.1]$ at 75.6% of the cost [69.6, 80.8], and 6.8 points above escalating the same share of runs at random [4.8, 8.7] (Figure 6b). The judge catches 85.4% of luna's failures and also escalates 40.9% of its successes; an oracle that escalates exactly the failed runs reaches 86.4% at $0.0131, so the gap between the judge and the oracle is 6.0 points.

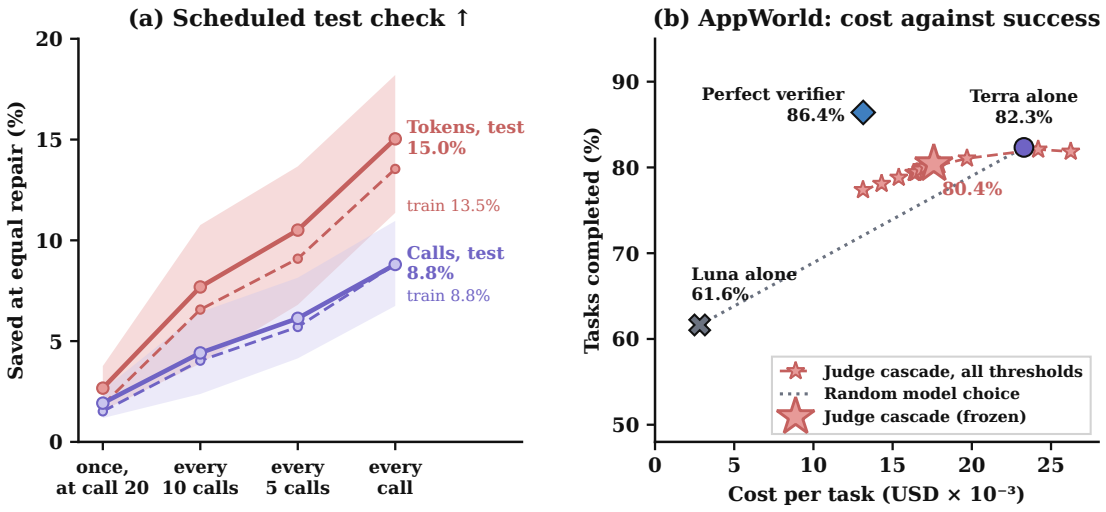


Figure 6: Cost savings from a scheduled check and from model escalation. (a) Savings in calls and tokens when the harness checks the visible tests on a schedule from call 20 and ends the run on a pass; bands are 95% intervals on the test set. (b) AppWorld: the judge-gated cascade at the rule fixed in advance (large star) and at twelve probability thresholds (small stars), against choosing a model at random (dotted line).

Scheduled Early Stopping. Many open runs already hold a correct patch that the agent has not yet submitted: the diff at call 20 is a repair for 14.2% of open training runs. A harness that runs the visible tests itself and ends the run on a pass recovers those calls. Replayed on the stored always-continue test runs, a check every five calls from call 20 saves 6.1% of calls [4.1, 8.1] and 10.5% of tokens [6.8, 13.7], and a check after every call saves 8.8% [6.7, 11.0] and 15.0% [11.4, 18.2], with the repair rate unchanged at 85.6% (Figure 6a). The training-set savings, computed first, agree at 5.7% and 9.1% for every five calls.

4.7 Diagnostic Analysis

Late Checkpoint. A restart at call 20 discards a short conversation, and the case for a restart is stronger late in a run, when the context is long. We therefore moved the decision to call 40 under an 80-call cap on the training repositories. Of 199 usable tasks, 67 are open at call 40; on those, continuing repairs 64.2% and resetting 49.3%, a difference of -14.9 points [-25.4, -1.7], and over all tasks the rates are 87.9% and 82.9%, a difference of -5.0 points [-10.0, -0.5] (Figure 4a). Six of the 67 continued runs end by exceeding the context window. Continuing thus still leads resetting when the decision comes late.

Cost-Penalty Sweep. Figure 4b sweeps the cost penalty as a secondary diagnostic. With a larger λ the controller stops some runs at call 20, and at the two budgets we interpolate its curve lies about 3 points above a uniform cap with the same mean calls, while its highest repair rate, 84.8%, stays below always continuing (Appendix C).

4.8 Case Study

The stored trajectories of the 14 call-40 runs that only CONTINUE repairs show where the calls go after a restart. These runs use 34.1 calls after the restart, against 22.2 for the continued conversation, and ten of them end on the call cap. Grouping shell commands by prefix labels 59% of post-restart commands as read-oriented, such as `cat`, `grep` and `git diff`, and ten of the 14 restarted runs end with edits to the same files as their continued counterparts. Seven pass more of the failing tests than the snapshot did, and one breaks a previously passing test. The digest tells the new conversation which files changed and what the last test printed. One restarted agent writes in its second message, "the current diff shows some changes were made, but they look like they might have been intentionally broken or are incorrect", and then

inspects the original files. Even on the 29 runs that both actions repair, the restart uses 24.4 calls after the checkpoint and the continuation 13.0.

5 Conclusion

In this paper, we evaluated three increasingly adaptive forms of harness-graph optimization for LLM agents under one out-of-sample rule. Within a single model, static graph search, pre-run routing and mid-run trajectory switching show no detectable gain in repair or task success over simple fixed baselines, and per-task advantages that appear in hindsight shrink on fresh runs. Other design choices produce large measured gains: a check whose executed outcome controls continuation, the diagnostics it returns, the call budget, selection and escalation across models, and stopping a run once a check passes. Within graph adaptation itself, the one measured benefit is in cost: the selected static graph uses 3.7% fewer tokens at a repair difference of $+0.2$ points. For coding agents with runnable tests, these results make context-preserving check-and-retry, scored on fresh runs, the baseline for future harness-graph optimizers.

Limitations

Our conclusions are bounded by the settings we tested. The prospective test has seven test repositories, so its intervals are coarse; the ordering of the fixed actions and the scheduled-check savings agree between the training and test sets, and the call-40 diagnostic on the training repositories shows the same ordering. The positive feedback findings use supplied tests: with the grading tests hidden, check-and-retry is only $+0.4$ points above the plain loop $[-3.0, 3.9]$ on the 232 held-out tasks. The experiments compare outcomes of whole subgraphs and do not isolate the loss of the conversation as the cause of the restart and fork deficits, and the four ablation variants do not give a unique causal split. The cascade and the scheduled check are replays of stored runs, and test time is not counted in their savings. Finally, the SWE-smith results use one 27B open model and the AppWorld results three graphs and two models; generalization to pools of specialized harnesses such as those of OO-Harness (Li et al., 2026) and to tasks without an automatic check remains open.

References

Ryan Ehrlich, Bradley Brown, Jordan Juravsky, Ronald Clark, Christopher Ré, and Azalia Mirhoseini. 2025. CodeMonkeys: Scaling test-time compute for software engineering. *Preprint*, arXiv:2501.14723.

Yuyuan Feng, Zhishang Xiang, Chaobin Yang, Qichao Ma, Zerui Chen, Yujing Zhang, Ke Huang, Chuanjie Wu, Zhaoxu Liu, Yili Wang, Xin He, Jiapu Wang, Zijin Hong, Hao Chen, Yuanchen Bei, Kun Wang, Shengyuan Chen, Ningyu Zhang, Enyan Dai, and 16 others. 2026. Graph engineering in the era of LLM agents: From individual intelligence to system intelligence. *Preprint*, arXiv:2608.21156.

Shengran Hu, Cong Lu, and Jeff Clune. 2024. Automated design of agentic systems. *Preprint*, arXiv:2408.08435.

Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. 2024. AI agents that matter. *Preprint*, arXiv:2407.01502.

Yoonho Lee, Roshen Nair, Qizheng Zhang, Kangwook Lee, Omar Khattab, and Chelsea Finn. 2026. Meta-Harness: End-to-end optimization of model harnesses. *Preprint*, arXiv:2603.28052.

Gaotang Li, Bhavana Dalvi Mishra, Yifei Ming, Yanfei Chen, Zhongying CuiZhu, Lesly Miculicich, Long T. Le, Rui Meng, Chun-Liang Li, Hanghang Tong, Chen-Yu Lee, and Tomas Pfister. 2026. Object-oriented harness: Evolving agent harnesses through object-oriented design. *Preprint*, arXiv:2610.28380.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. 2024. SWT-Bench: Testing and validating real-world bug-fixes with code agents. *Preprint*, arXiv:2406.12952.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Preprint*, arXiv:2303.11366.

Srinath Srinivasan, Lunxiao Li, Tim Menzies, and Wesley Assunção. 2026. Continue or restart? the limits of early failure prediction in coding agents. *Preprint*, arXiv:2610.35971.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. *Preprint*, arXiv:2407.18901.

Chenyu Wang, Yunbo Lyu, Junda He, Zhou Yang, Chenxing Zhong, Yaniv Harel, and David Lo. 2026. Fail-fast, restart-smart: Early failure prediction and restart for SWE agentic tasks. *Preprint*, arXiv:2608.03222.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying LLM-based software engineering agents. *Preprint*, arXiv:2407.01489.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2025. SWE-smith: Scaling data for software engineering agents. *Preprint*, arXiv:2504.21798.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. ReAct: Synergizing reasoning and acting in language models. *Preprint*, arXiv:2210.03629. ICLR camera-ready version.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2024a. AFlow: Automating agentic workflow generation. *Preprint*, arXiv:2410.10762.

Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, et al. 2024b. Diversity empowers intelligence: Integrating expertise of software engineering agents. *Preprint*, arXiv:2408.07060.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. 2024. Language agents as optimizable graphs. *Preprint*, arXiv:2402.16823. In Forty-first International Conference on Machine Learning (2024).

A Protocol of the Mid-Run Test

Subgraphs and Features. The eight features at call 20 are: whether a non-test source file was edited, the diff size, the number of test runs, the latest visible pass ratio and its change since the first test run, the share of repeated commands in the last ten steps, the length of the current streak of identical errors, and the tokens used. The reset digest contains `git status`, the diff summary and the end of the latest test output, built to a 2,000-character target. FORK-JOIN runs branch A and then branch B with 17 calls each, and a selector without model calls prefers the branch that passes all visible tests, then the higher pass ratio, then the smaller diff. The cost c_g is the predicted number of extra calls divided by 34.

Endpoints. Endpoint A compares the controller at λ^* with a uniform call cap matching its mean calls and with random routing with the same action shares, averaged over 100 seeds; it requires both differences to be at least $+2.0$ points with lower interval ends above zero. Endpoint B compares the controller at λ_{cons} with always continuing; it requires the lower end of the repair difference to be above -1.5 points and a saving in calls or tokens of at least 20% with a lower end above 10%. Intervals come from a paired bootstrap over repositories with 5,000 draws.

Operating Points. Cross-validation on the training set found no penalty with mean calls in the registered window of 34 to 38, since always continuing averages 33.0, so $\lambda^* = 0.0$, the value closest to it. No penalty stayed within 1.0 point of always continuing, so $\lambda_{\text{cons}} = 0.01$, the value with the highest cross-validated repair. Both fallbacks were recorded before the test. FORK-JOIN was kept in the action set; its largest cross-validated share was 3.5%.

Gate	Scope	Verdict
1	Split: 200 and 250 tasks, no repository in both, no test repository used in earlier tuning, file hashed	PASS
2	Five-task smoke run: features read calls 1 to 20 only; reset keeps the edits; fork branch B starts from the clean repository; selector uses no model call	PASS (second round)
3	Controller and configuration saved and hashed; no test repository touched during training	PASS
4	Evaluation uses the Gate 3 hashes; 5,000-draw paired bootstrap over repositories; mean calls of the cap baseline and action shares of the random baseline matched	FAIL on cap matching only

Table 3: The four audited protocol gates.

Audit Record. The audit verified the hashes of the controller and configuration, the split, the pairing of branches, the 5,000-draw repository bootstrap and the reproduction of every number. It recorded Gate 4 as failed on one item: the uniform-cap baseline cannot be matched to the controller's budget. The controller averages 31.27 calls per task and always continuing under the 54-call cap averages 30.99, so no cap up to 54 reaches the controller's spending, and the baseline used is the largest cap, which is always continuing itself. Always continuing therefore has the higher repair rate and the lower call count at once, 85.6% at 30.99 calls against 84.0% at 31.27, while the controller uses 2.3% fewer tokens. Gate 2 failed once before any training run, when the check that branch B starts from a clean repository was corrected to compare against the initial commit. In the first round of Gate 4 the interval of the random-routing comparison resampled a fixed baseline; the shares are now re-matched inside every bootstrap draw, which moved the interval from $[-2.0, 0.5]$ to $[-1.3, 0.1]$.

Deviations and Exclusions. The reset digest uses a 2,000-character target and enforces neither a hard character limit nor the protocol's 500-token limit. Early-stopped runs are graded on their checkpoint patch rather than scored as automatic failures, the same way a uniform cap of 20 calls is graded; the decision rule still scores a stop as zero repair at zero cost. The graders of the seven test repositories had been verified with reference patches, without a model, before the protocol was fixed. Six training tasks of one repository, python-dotenv, fail before any branch runs, because a test of that repository creates an unreadable file and the snapshot copy stops on it; the 194 usable tasks cover all 15 repositories. The call-40 diagnostic consists of new runs with their own snapshots, in which the same failure recurs for the one python-dotenv run still open at call 40; the other twelve runs of that repository finish earlier and need no snapshot. Cross-validation uses five folds grouped by repository.

B Static Search Space and Procedures

Design Variables. The nine design variables are: plan first (no, yes); reproduce the bug first (no, yes); offer the checker as a tool (no, yes); review before submitting (none, same conversation, after summarizing); enforce the test check (no, yes); after a failed check (back to editing; find the cause, then edit; find the cause, start over); retries (1, 2, 3); summarize a long conversation (never, past 16k tokens, past 32k tokens); and after 20 steps without an edit (nothing, remind, find the cause, start over, summarize). The settings for a failed check and for retries matter only with the enforced check, which leaves $24 \times 10 \times 15 = 3{,}600$ distinct graphs.

Search Procedures. The table scores 32 distinct graphs on 188 tasks: the two anchors and 30 sampled ones, 16 with the enforced check and 16 without, with the other design variables drawn uniformly and duplicates rejected. Four allocation procedures are replayed on it: uniform allocation, successive halving, Thompson sampling, and Thompson sampling with a linear model over the design variables. In a live local search, the current graph and a neighbour that differs in one design variable run on the same 16 tasks, and the neighbour replaces the current graph when it repairs more of them, so a search of 512 runs takes 16 steps. Five searches choose an untried neighbour at random, and five try prescribed changes in descending order of the current graph's failure-reason counts (incorrect patch submitted, run ended on a failed check, no edit, call cap, context limit), falling back to a random untried neighbour.

Pre-Run Router. The ridge penalty is 3.0 with the intercept unpenalized, the features are standardized, and the word-count feature is $\log(1 + \text{issue word count})$. The training data are 376 units, the 188 search tasks under two test conditions, with one scored run per graph.

C Additional Diagnostics

Per-Task Choice on AppWorld. The choice is the graph with the most successes on the two known runs; a tie goes to the fixed graph when it is among the best and otherwise to the first graph in the order incremental, plan-then-execute, whole-program-then-rewrite. Per-run success for the three graphs is 61.6%, 57.9% and 20.0% with gpt-6-luna and 82.3%, 72.0% and 36.1% with gpt-6.1-terra. Among 32 graphs on the 188 SWE-smith search tasks, the best single graph repairs 82.4% and the per-task best in hindsight 92.0%. The SWE-smith fresh-run comparison of Figure 1 is descriptive and has no interval. The 417 AppWorld tasks form 139 scenarios, which are the bootstrap clusters.

Cascade Details. The judge returns a verdict together with a probability from 0 to 100 that the task was completed. The rule fixed in advance uses the verdict alone and escalates 58% of runs; the twelve small stars in Figure 6b escalate when the stated probability is below a threshold and are a description only. The cascade cost includes the luna run, the judge call at \$0.0004 per run, and the terra run when it is started. The random-escalation baseline matches the escalation rate rather than total dollar cost. The oracle cascade costs 56% as much as terra alone, and the judge cascade costs \$0.0045 more per task than the oracle.

Cost-Penalty Sweep. As λ grows, the controller stops choosing RESET-WITH-DIGEST and FORK-JOIN and starts choosing EARLY-STOP for some runs. Interpolated to equal mean calls, its curve gives 66.1% against 62.9% for a uniform cap at 24 calls and 78.4% against 75.6% at 28, differences of 3.2 and 2.8 points. The sweep is interpolated and has no interval.

Scheduled Check and Budget Extension. One forced check at call 20 ends 37 of the 250 test runs and saves 1.9% of calls [1.2, 2.6]; on the training set a check after every call saves 8.8% of calls and 13.5% of tokens. The replay uses the same tests for ending a run and for grading it. The 100-call experiment compares two checkpoints within the same trajectory, so it does not directly contrast continuation with restarting. In the call-40 diagnostic, both actions succeed on 29 open runs, only continuing on 14 and only resetting on 4; among the 61 continued runs that do not exceed the context window, resetting still has the lower repair rate. Of the 14 restarts paired with a successful continuation, ten end on the call cap and four on the node-visit limit, each with a submission as its final message.

Ablation and Withheld Tests. The 200 ablation tasks are sampled from the 900 fresh tasks, 13 or 14 per repository, and the budget extension uses the same 200 tasks. The full gate leads forced continuation by 22.0 points. On the withheld-test endpoint, check-and-retry is better on 43 tasks and worse on 15; the withheld-test rejection rate was not measured for mid-run checkpoints. On the mid-run test set, always resetting uses 19% fewer tokens than always continuing, at the cost of 3.9 extra calls and 14.0 fewer repair points. The trajectory totals count each shared run once and each reset and fork branch separately.

Artifacts. The split, the frozen controller, its configuration and the gate record are in `artifacts/`, the result summaries behind every number in `results/summary/`, and the figure script in `scripts/make_report_figures.py`.