Learning Repair Utility for Coding Agents

Small Decision Models for Verification and Recovery

Anonymous Authors

Working draft • 23 September 2026 • Utility training stopped after diagnostic gate

Abstract. Choosing the next action after a coding agent produces a patch requires distinguishing current correctness from the outcome of further work. We investigate whether matched executions can train a compact, non-generative controller under a fixed repair model. A three-seed CPU study trains a 0.6B current-correctness predictor, reaching 77.1% mean accuracy on 32 development states from eight tasks. A 108-execution pilot on 12 final-candidate training states yields no repair of the three initially unsuccessful candidates; capped restart frequently damages correct patches. In a separately frozen diagnostic enriched for 12 initially unsuccessful training tasks, 83 of 84 executions are judged. Public-test feedback repairs one task at both seeds, but no pair of actions exhibits replicated opposite preferences across tasks. The prespecified feasibility gate fails. These results establish an executable collection protocol but do not establish sufficiently task-diverse intervention gains for the planned utility training. An action-utility model and independent test-set evaluation remain outstanding.

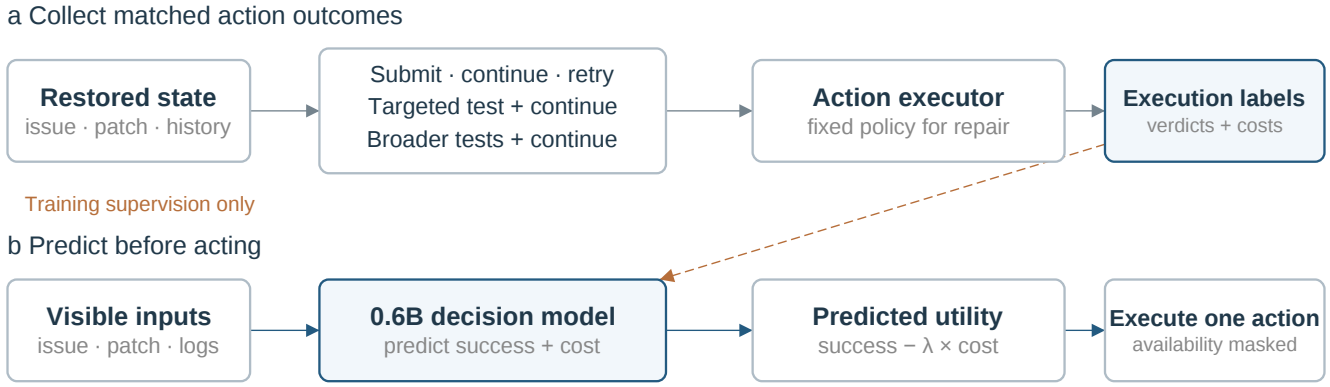


Figure 1. Repair-utility learning and deployment. Matched executions supply verdicts and costs as training targets. Submit returns the patch; retry resets code and history. At decision time, the proposed small model would use visible state to predict action success and cost. Dashed supervision would be available only during training. The diagnostic gate stopped utility training and controller evaluation for this action menu.

1 Introduction

A coding agent that has produced a patch still faces a consequential decision: submit it, inspect another test result, keep editing, or start over. These actions spend different resources and expose the agent to different information. A useful controller must predict what an action will accomplish, rather than treating every uncertain patch as a reason to spend more compute.

Patch correctness is an incomplete target for this decision. Consider two patches with the same probability of passing the official evaluator. The first has a localized error that a short regression test can expose. The second depends on a mistaken interpretation of the issue, so repeating nearby tests supplies little useful information. Their current correctness probabilities can be equal even when the best next actions differ. Conversely, a test that reliably reports a failure need not provide feedback that the repair model can use. The value of verification depends on the code state, the diagnostic, and the model that receives it.

Prior work has established several components of this problem. Lightweight monitors can predict failing software-agent trajectories and trigger restarts (Wang et al., 2026). Bayesian controllers choose criticism, refinement, verification, and stopping using cost-sensitive beliefs (Papamarkou et al., 2026). Execution-free reward models distinguish candidate patches, while learned testing agents improve the feedback available to a repair agent (Shum et al., 2025; Tao et al., 2026). These results make a stronger question possible: does directly learning the outcome of each intervention improve control over learning current correctness and then applying an action rule?

Figure 1 summarizes the proposed pipeline. We formulate the decision as supervised action-value estimation at a repair boundary. A fixed generator produces a candidate patch. A small controller then chooses one of five bounded actions. During data collection, each nonterminal action is actually executed from the same restored candidate state, with a fresh environment and a prescribed random seed. An official evaluator labels only the final patch. The decision model learns a separate success probability and cost prediction for each action. The probabilities are not a categorical distribution: several actions may succeed, or all may fail.

This design keeps the learning problem narrow. The repair model remains fixed, the test menu is determined before action execution, and all continuation actions receive the same token and tool-call limits. The comparison can therefore isolate the supervisory target without attributing changes in the repair model or additional sampling to the controller. A smaller classification error is useful only if the resulting choices improve the task-level success–cost relationship.

The implemented contribution consists of an executable same-state intervention protocol, a non-generative action-outcome model, and a matched evaluation that separates verification

choice from recovery control. The initial evidence establishes the execution pipeline: 12 candidate verdicts are reproduced exactly, and 24 public-test calls complete. The controlled outcome comparison is the decisive experiment, because infrastructure correctness and useful agent intervention answer different questions.

2 Related Work

Direct decisions and lightweight classifiers. Typed outputs are an interface for integrating uncertain judgments into software. A binary or categorical prediction can be read directly from a classifier instead of generated as a string. GLiClass supports lightweight classification with flexible label descriptions and also studies reinforcement learning for classification (Stepanov et al., 2025). This line of work establishes that direct probabilistic outputs and small models are not sufficient novelty by themselves. Our model uses a conventional open backbone and classification heads. Its research target is the supervision supplied by executed repair actions, rather than the output schema or the removal of JSON generation.

Trajectory monitoring and agent control. Fail-Fast, Restart-Smart trains a small monitor on observable software-agent prefixes and execution-derived supervision, then combines termination with an edit-preserving restart (Wang et al., 2026). This is a close comparator for any small-model recovery controller. Our first setting places a single decision after a candidate patch has been produced and evaluates a fixed menu of interventions. It distinguishes a continuation that retains the candidate and history from a restart that returns to the original repository. Vasudev et al. (2026) directly distinguish accurate failure prediction from useful intervention through disruption and recovery effects. Our empirical question is whether action-conditioned outcomes provide information beyond a current-correctness predictor and a recovery rule.

Cost-sensitive verification. Bayesian Control formulates coding-agent orchestration as a sequential hypothesis-testing problem, maintaining a belief over correctness and choosing among critics, refinement, verification, and stopping (Papamarkou et al., 2026). It already provides a decision-theoretic account of costly verification and includes repository-level tasks. We build on that motivation. The distinguishing candidate contribution is a learned mapping from the concrete state and available diagnostic to the outcome of a bounded repair intervention. An empirical transition-kernel baseline tests whether a global model of repair and breakage is sufficient. That baseline is identified as a one-step adaptation, rather than a reproduction of the original dynamic program.

Patch reward models and learned feedback. SWE-RM studies execution-free feedback and separates ranking performance from classification accuracy and calibration (Shum et al., 2025). The distinction matters here: a model that ranks patches well does not automatically identify which action to take on a particular patch. ExecCritic separates test construction from repair, qualifies generated tests, and trains the two roles independently (Tao et al., 2026). Our first experiment fixes the repair model and selects from existing public tests; it does not train a test generator. Both approaches require careful treatment of test validity, because an apparently informative failure can point the agent toward an incorrect behavioral target.

Test selection and evidence validity. Predictive Test Selection learns which tests are likely to expose regressions from historical changes and execution outcomes (Machalica et al., 2019). That objective differs from predicting how useful a test's feedback will be to a subsequent repair model. A failure-probability selector would test the narrower hypothesis that feedback value is explained entirely by failure detection. The first matched comparison instead isolates action-outcome supervision against current correctness; a direct failure-discovery comparator requires additional public-test labels and is reserved for the extended study. Work on state-bound evidence further shows why test results must refer to the exact program state on which they were obtained, and why continued revision can destroy a correct solution (Gao et al., 2026). We adopt exact-state restoration and fresh execution environments as experimental requirements. Their role is to support a valid comparison, not to serve as a separate architectural contribution.

3 Problem and Data

Decision state. Let x_i denote a repair state for task i. The controller observes an issue description, a patch relative to the initial repository, recent tool observations, a public-test menu, and the remaining repair budget. The underlying executor can also restore the candidate's complete saved history. The controller's input is bounded to 4,096 tokens through a fixed, label-independent allocation across fields. Future observations, reference patches, and official evaluation outcomes are excluded from this input.

Actions. The action set is

$$\mathcal{A} = \{\texttt{submit}, \texttt{continue}, \texttt{retry}, \texttt{test_targeted}, \texttt{test_regression}\}.$$

Submission returns the current patch. Continuation restores the candidate and its visible history. The two testing actions run a predetermined public check on the restored candidate, add its actual output to the history, and then continue repair. Restart restores the original repository and begins a fresh attempt without the earlier reasoning history. A testing action is unavailable when its menu has no legal candidates.

All nonterminal actions receive at most 8,192 new output tokens, 30 tool calls, and 1,200 seconds for repair. A prescribed pretest receives an additional 300-second ceiling, and its time is charged to the action. The limits specify a common operating regime rather than equal realized cost: an action that submits early can use less than its allowance. The first experiment makes one controller decision per candidate, so it estimates bounded intervention value rather than an unrestricted long-horizon policy.

Public tests and official labels. The test menu is constructed from tests already present in the initial repository. A deterministic path-association rule selects up to five targeted test functions and up to twenty broader test functions. Candidate-modified test files are excluded. Pytest, Django's native runner, and SymPy's native runner execute the declared selections. The controller receives the pretest output only after choosing that

action. The official hidden test patch is introduced only for final grading and is never fed back to the repair model.

Frozen cohorts. The available clean pilot contains 320 trajectories from 40 tasks in eight repositories. We select four candidates per task with a seeded hash of the source trial identifier, before reading the outcome label. One task from each repository forms the development set, leaving 32 training tasks and eight development tasks: 128 and 32 candidate states, respectively. All candidates from a task remain in one split. The independent test cohort consists of 60 frozen task identifiers selected after excluding the existing 150-task pool and a separate 100-task holdout used by another study. Two fresh candidates per test task are planned after development configuration freeze.

Restoration and missingness. We export final patches and visible histories; the protocol does not depend on retaining every intermediate Git object. Each action runs in a fresh environment. Setup removes access to later Git history, and the agent phase is network-isolated. Binary patch application and the initial repository commit are checked before the intervention. Transport, setup, and grading failures retain an unavailable outcome instead of receiving a failure label. When an action lacks the required pair of judged seeds, its success target is masked. Complete frozen-cohort coverage is required for the headline comparison.

4 Learning Repair Utility

Action outcomes. Fix a repair policy π, its execution scaffold, and a repair budget B. For a candidate x, action a, and sampling seed r, let $Y(x, a, r) \in \{0, 1\}$ denote the official final outcome. Define

$$q_a(x) = \Pr\big(Y(x, a, R) = 1 \mid x, \pi, B\big), \tag{1}$$

$$c_a(x) = \mathbb{E}\big[C(x, a, R) \mid x, \pi, B\big]. \tag{2}$$

For submission, Y is the current patch's verdict. For another action, Y is the verdict after its actual bounded execution. This distinction prevents the logged terminal outcome of an unrelated trajectory from being used as the label for an intervention that was never performed.

For each available nonterminal action, the collection procedure restores the same candidate, executes the action with two prescribed seeds, and records the outcome and resource measurements. Restart intentionally changes the starting code to the pristine repository. Reusing seed identifiers across actions controls the sampling configuration; it does not make different prompts consume identical random draws or eliminate stochastic variance.

Model. We adapt Qwen3-0.6B (Yang et al., 2025) with rank-16 low-rank updates to its attention projections. The hidden state of the final nonpadding input token, $h_\theta(x)$, feeds two linear heads. The first returns five logits $z_a(x)$ with probabilities $\widehat{q}_a(x) = \sigma(z_a(x))$. The second returns nonnegative log-cost estimates

$$\widehat{\ell}_a(x) = \mathrm{softplus}(v_a^\top h_\theta(x) + b_a), \tag{3}$$

$$\widehat{c}_a(x) = \exp(\widehat{\ell}_a(x)) - 1. \tag{4}$$

The output is produced in one forward pass without decoding answer tokens. The five probabilities are separate Bernoulli estimates, not a softmax over actions. A state can be repairable by several actions, and high success probability does not require the other actions to receive low probability.

Supervision. For each candidate and action, $\bar{y}_{ia}$ averages the judged outcomes across the prescribed seed pair, and $\bar{c}_{ia}$ averages the measured normalized cost. Submission has one outcome. Let m_{ia} indicate a complete correctness target and u_{ia} indicate an available cost target. The implemented loss is

$$\mathcal{L}_i = \frac{\sum_a m_{ia}\ \mathrm{BCE}(z_{ia}, \bar{y}_{ia})}{\max(1, \sum_a m_{ia})} + \frac{\sum_a u_{ia}\ \mathrm{Huber}(\widehat{\ell}_{ia}, \log(1 + \bar{c}_{ia}))}{\max(1, \sum_a u_{ia})}. \tag{5}$$

We average over states with usable supervision. Outcome masking and cost masking are separate: a failed grader can make correctness unavailable even when the resources consumed by the action are known. The default optimizer uses learning rate 10^{-4}, effective batch size 32, and at most five epochs. Development predictive loss selects the checkpoint; three training seeds are specified for the main comparison.

Cost and action choice. The initial policy objective normalizes output generation and public-test service time as

$$C = \frac{N_{\mathrm{new}}}{8192} + \frac{T_{\mathrm{pretest}}}{300}. \tag{6}$$

This quantity is a control objective, not a dollar price or a measure of total compute. Raw measurements separately retain input tokens, output tokens, public-test CPU time, repair time, available repair CPU time, setup time, final-grading time, and decision latency. Candidate-generation cost is shared by the compared downstream choices and is reported for the full workflow.

The controller selects

$$a^*(x) = \arg\max_{a \in \mathcal{A}(x)} \big[\widehat{q}_a(x) - \lambda \widehat{c}_a(x)\big]. \tag{7}$$

The available-action mask is applied before selection. A fixed grid $\lambda \in \{0, 0.02, 0.05, 0.1, 0.2\}$ is evaluated on development data. The selection rule seeks the lowest observed-cost operating point whose observed development success lies within two percentage points of the best eligible fixed policy or supplied frozen reference policy. If no point satisfies the tolerance, the implementation retains $\lambda = 0$ and records that the development gate was not met. The resulting threshold is frozen before test predictions. This development rule does not establish a distribution-free risk guarantee.

Correctness-based comparator. To isolate the learning target, a matched model predicts only whether the current patch passes. A training-only empirical transition kernel $K_a(y) = \Pr(Y_a = 1 \mid Y_{\mathrm{submit}} = y)$ maps this estimate to action success:

$$\widehat{q}_a^{\mathrm{corr}}(x) = (1 - \widehat{p}(x))K_a(0) + \widehat{p}(x)K_a(1). \tag{8}$$

The submission entries are fixed to $K_{\mathrm{submit}}(0) = 0$ and $K_{\mathrm{submit}}(1) = 1$. Nonterminal entries use a symmetric beta

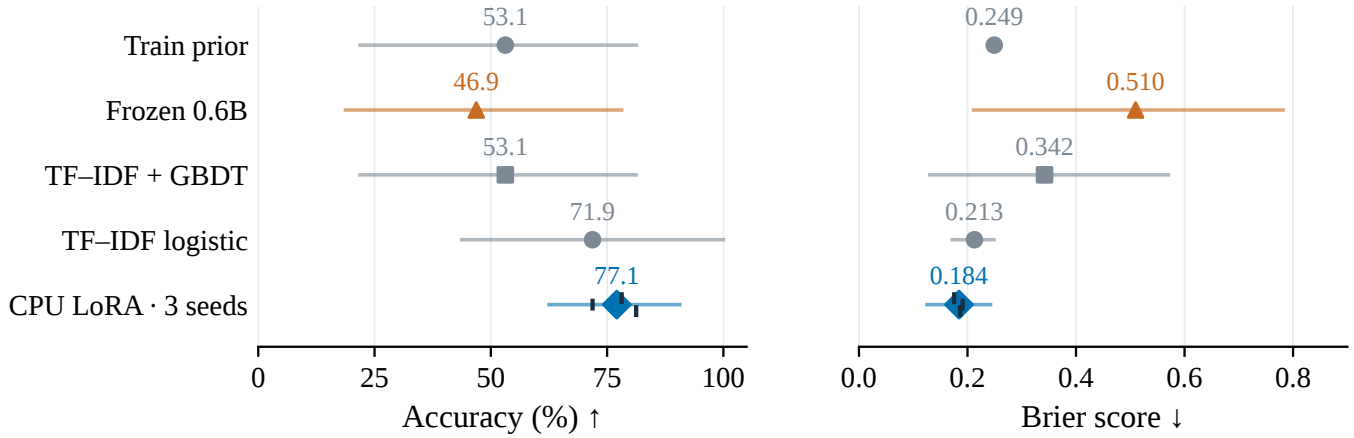


Figure 2. Current-correctness development comparison. Points are means over 32 states from eight tasks; horizontal bars are 95% task-bootstrap intervals (5,000 resamples), conditional on the trained models. The LoRA point averages each metric across three completed seeds, whose individual values are shown by small ticks. The frozen model is zero-shot; the other predictors use 128 training states. These are descriptive development results, not action-utility or independent test results.

Table 1. Current-correctness development results. LoRA reports the mean across three seeds; other rows are single fitted or frozen baselines. All rows evaluate the same 32 states.

Method	Acc. ↑	Brier ↓	AUC ↑
Train prior	.531	.249	.500
Frozen 0.6B	.469	.510	.522
TF–IDF + GBDT	.531	.342	.494
TF–IDF logistic	.719	.213	.804
CPU LoRA (3 seeds)	**.771**	**.184**	**.846**

prior with pseudocounts $(0.5, 0.5)$. Both controllers use the same action menu and cost-selection rule. The action-outcome model can condition repairability on the complete observed state, whereas the comparator compresses that dependence into current correctness and an aggregate recovery kernel. Their paired comparison is the primary test of the method.

5 Preliminary Evidence and Evaluation

Completed CPU study. The three current-correctness LoRA runs completed three epochs each using eight CPU threads and no GPU allocation. They train on the same 128 states and evaluate on the same 32 development states from eight tasks. Each run took 1.91–2.05 process hours. Their final accuracies are 78.1%, 71.9%, and 81.3%; mean accuracy is 77.1% with a sample standard deviation of 4.8 percentage points. These checkpoints predict current correctness, not action utility.

Figure 2 shows descriptive task-cluster intervals and individual LoRA seed estimates. The mean Brier reduction relative to logistic regression is 0.0285, with a paired task-bootstrap 95% interval of $[-0.0639, 0.0077]$ for LoRA minus logistic. The interval crosses zero. These development results motivate reusing the correctness model, but do not establish a reliable advantage on new tasks or an improvement in agent control. Full CPU learning curves are given in Appendix B.

Execution integrity. All 160 training and development states have public-test menus. The initial 12-state audit reproduces every original official patch verdict. Its 24 public-test calls complete: 22 pass and two return test failures. Two states pass both selected public checks while failing the official evaluator. Public feedback therefore remains separate from the correctness label. The audit's median public-test service time is 1.728 seconds.

Table 2. G001 final-state pilot. Resolution is the fraction of officially successful executions. Repair counts condition on the three initially unsuccessful candidates (two seeds each). Token means count added outputs only.

Action	Success	Res. %	Repairs	Tokens
Submit	9/12	75.0	–	0
Continue	18/24	75.0	0/6	1,159
Targeted	17/24	70.8	0/6	607
Regression	18/24	75.0	0/6	636
Retry	3/24	12.5	0/6	5,999

Controller overhead. A CPU forward pass of the 0.6B architecture takes 0.728, 3.374, and 8.788 seconds at 512, 2,048, and 4,096 input tokens, respectively. These are three-repeat medians on eight CPU threads in FP32. Thus a long-input controller can cost more wall time than a selected public test. Any efficiency claim must include decision overhead and should be tested against avoided repair or retry work, not inferred from fewer test calls. The complete hardware protocol appears in Appendix B.

Completed final-state action pilot. The frozen 12-state pilot contains 11 training tasks. All 108 executions receive official verdicts: one submission check per state and two seeds for each of four nonterminal actions. Non-restart initial trees match the corresponding submitted candidate, and no selected public test is modified. Table 2 reports descriptive branch counts, not a learned-controller comparison.

No action repairs any of the three initially unsuccessful states. Continuation and regression feedback preserve all observed submission outcomes, while targeted feedback damages one successful state in one seed. Restart damages 15 of 18 initially successful branches. All 24 restarts reach a budget limit: 16 hit the tool-call cap and eight hit the output-token cap. This

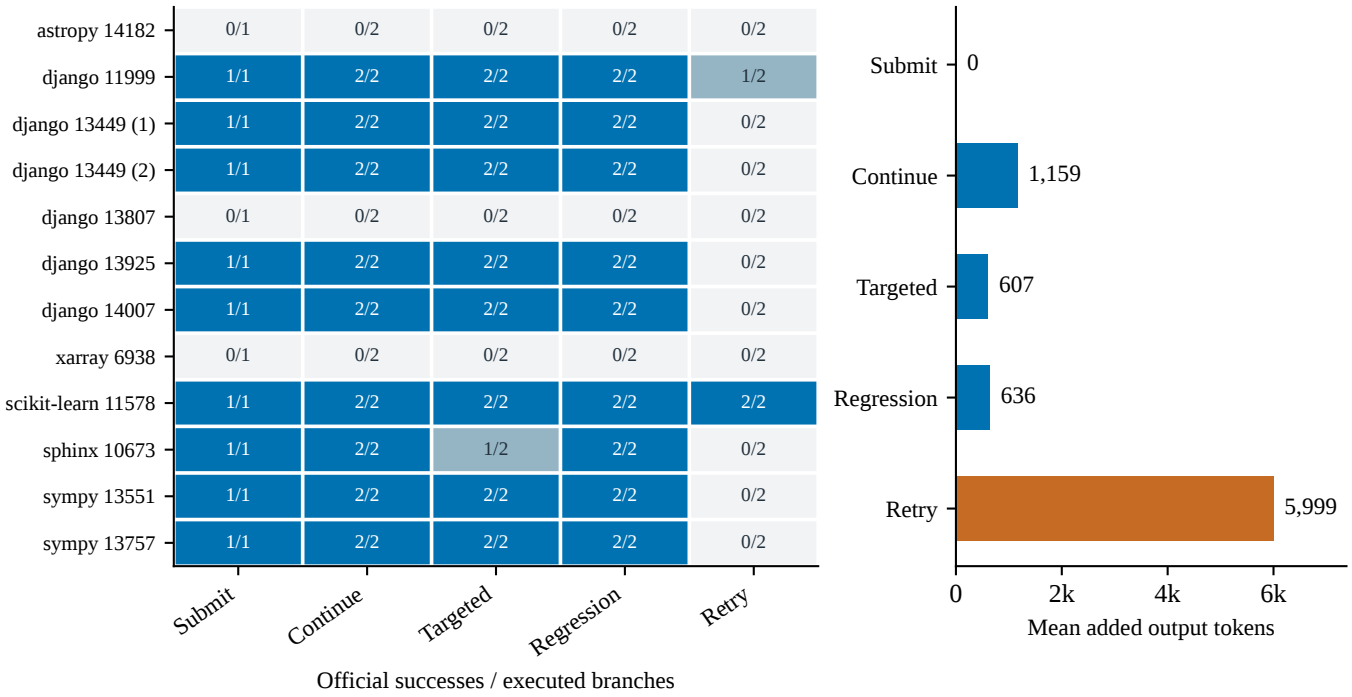


Figure 3. Observed outcomes and added generation in G001. Left: official successes/executed branches for 12 candidates from 11 training tasks; the two Django 13449 rows are different candidates. Submission is evaluated once, other actions at two seeds. Right: candidate-weighted output-token means over 12 submission evaluations or 24 nonterminal branches, excluding initial generation, prefill, tests, grading, and decision overhead. All repair actions use the frozen 9B model and the common 8,192-token/30-tool/1,200-second limits, with up to 300 additional seconds for a pretest. None of the three initially unsuccessful candidates is repaired. Repeated candidates and seeds are not independent tasks.

characterizes the specified short restart policy, not unrestricted restart quality. Two long-history states also produce context-limited branches. The observed per-state oracle cannot improve on always submitting in this cohort, so action disagreement alone is insufficient evidence for state-dependent control.

The completed G001 allocations consumed 1.805 A100-hours, including startup, idle time, smoke runs, and earlier infrastructure attempts; this is collection cost, not deployment savings.

Frozen failure-enriched diagnostic. Before observing any new action outcomes, we selected one originally unsuccessful final candidate from each of 12 previously unbranched training tasks. A fixed hash selected tasks and candidates after a prespecified context-fit check; no development or test task was used. We evaluated submission, continuation, targeted public feedback, and broader public feedback, with two seeds for each nonterminal action. Of 84 frozen records, 83 receive an official verdict. The remaining targeted-feedback branch has a transport read error and retains an unavailable outcome. All 12 submitted patches fail as expected.

The four successful repair executions arise from a single Django task: targeted and broader feedback each repair it at both seeds. No other judged branch repairs another task, and no pair of non-restart actions has replicated opposite preferences across different tasks. The prespecified gate required at least 95% judged coverage, three distinct tasks with replicated repair, and a replicated action crossover. Coverage is 83/84 and the baseline verdicts match, but the repair and crossover criteria fail. F1 consumed 0.783 additional A100-hours. We therefore

Table 3. Failure-enriched F1 diagnostic. Only originally unsuccessful training final states are included. Each successful test arm comprises the same task at two seeds; the one unavailable targeted outcome is excluded from its success denominator. These fractions are conditional feasibility observations, not deployment accuracy.

Action	Judged	Successful
Submit	12/12	0
Continue	24/24	0
Targeted feedback	23/24	2
Broader feedback	24/24	2

stop full final-state branch expansion and action-utility model training under this protocol. The existing development and 60-task test cohorts remain sealed for action selection.

Baselines and attribution. The originally planned full comparison would include fixed policies, a CPU gradient-boosted predictor, direct action heads, and current-correctness prediction with empirical action-transition kernels. Those action-utility models were not trained: the feasibility gate stopped the expansion before a policy comparison. The completed evidence is therefore a matched-action diagnostic, not a controller benchmark.

Statistical scope. Tasks, rather than branch seeds, are the unit for any future paired interval. G001 covers 11 tasks and F1 is deliberately enriched for failed final states. The two-percentage-point tolerance and 15% saving target were engineering criteria, not achieved results. One F1 outcome is unavailable, and the 60-task test cohort remains sealed.

6 Limitations and Conclusion

The completed measurements establish executable restoration and a trained current-correctness predictor on the development cohort. G001 contains only three initially unsuccessful candidates and observes no repairs; its action differences largely reflect damage from budget-limited restart. The failure-enriched F1 diagnostic finds replicated repair in one task but no cross-task action crossover. Under the fixed final-state policy, these results do not support an action-conditioned success–cost advantage or justify fitting a more complex controller. The 32 development states represent eight task clusters, limiting threshold selection and baseline comparisons despite repetition over three training seeds.

The first controller makes one decision after a candidate patch is produced and uses one fixed repair model. It does not establish performance for repeated decisions along an entire trajectory or transfer across repair models. Its success probabilities are conditioned on a particular scaffold and budget; changing either can change calibration. The public-test menu is based on static path association, so inadequate coverage or misleading feedback can limit every selector built on that menu.

Data collection executes several actions per training state. Any deployment cost advantage must therefore be reported together with the cost of collecting labels and training the controller. CPU memory and timing measurements cannot determine GPU throughput, and low-rank adaptation reduces trainable parameters without eliminating the activation memory needed to process long inputs.

References

Xueping Gao, Jianwei Yang, and Qiang Yang. Looping is not reliability: State-bound evidence and typed revision contracts for agentic code repair. arXiv preprint, 2026. URL https://arxiv.org/abs/2607.24604.

Mateusz Machalica, Alex Samylkin, Meredith Porth, and Satish Chandra. Predictive test selection. arXiv preprint, 2019. URL https://arxiv.org/abs/1810.05286.

Theodore Papamarkou, Vladislav Smirnov, Viktor Mazanov, Artem Vazhentsev, Preslav Nakov, Timothy Baldwin, and Artem Shelmanov. Bayesian control for coding agents. arXiv preprint, 2026. URL https://arxiv.org/abs/2606.24453.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, X. W., Jiaxi Yang, Yuzhen Huang, Junyang Lin, and Junxian He. Swe-rm: Execution-free feedback for software engineering agents. arXiv preprint, 2025. URL https://arxiv.org/abs/2512.21919.

Ihor Stepanov, Mykhailo Shtopko, Dmytro Vodianytskyi, Oleksandr Lukashov, Alexander Yavorskyi, and Mykyta Yaroshenko. Gliclass: Generalist lightweight model for sequence classification tasks. arXiv preprint, 2025. URL https://arxiv.org/abs/2508.07662.

Leitian Tao, Baolin Peng, Haorui Wang, Hang Wang, Hao Cheng, Wenlin Yao, Qianhui Wu, Tao Ge, Sharon Li, and Jianfeng Gao. Execcritic: Learn to test, test to improve for coding agents. arXiv preprint, 2026. URL https://arxiv.org/abs/2609.09133.

Rakshith Vasudev, Melisa Russak, Dan Bikel, and Waseem Alshikh. Accurate failure prediction in agents does not imply effective failure prevention. arXiv preprint, 2026. URL https://arxiv.org/abs/2602.03338.

Chenyu Wang, Yunbo Lyu, Junda He, Zhou Yang, Chenxing Zhong, Yaniv Harel, and David Lo. Fail-fast, restart-smart: Early failure prediction and restart for swe agentic tasks. arXiv preprint, 2026. URL https://arxiv.org/abs/2608.03222.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report. arXiv preprint, 2025. URL https://arxiv.org/abs/2505.09388.

Appendix: implementation and CPU evidence

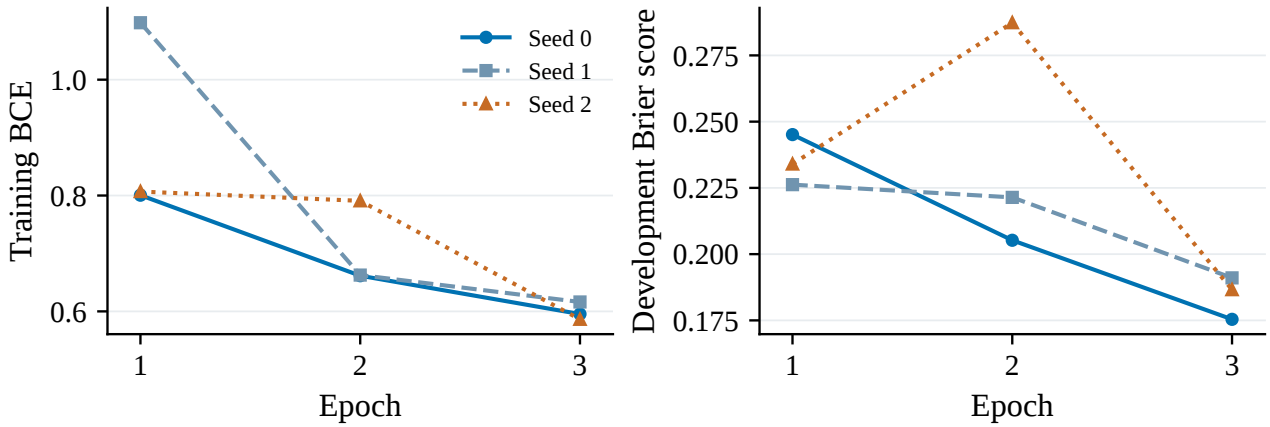


Figure 4. Complete CPU training histories. Each run uses 128 current-correctness training states and the same 32 development states. The three lines are training seeds, not independent test cohorts. Development Brier is non-monotone for some seeds. These are not action-utility learning curves.

A Implementation

Restoring history. Serialized tool arguments are converted into the objects required by the chat template. Historical reasoning text is preserved explicitly inside assistant messages because the template otherwise strips it after a new user message. This preserves text, not the original token stream. Overlength histories are recorded rather than silently truncated. Retry resets both code and history.

Verifier state. The executor records the starting tree and exports the final patch. Changes to selected public tests are flagged. Before grading, the Git index is reset while preserving the worktree, because some official verifiers depend on staged/untracked distinctions. The agent phase remains network-isolated.

Input boundary. Only issue text, patch text, existing public observations, test descriptions, and known budgets enter the controller. Source-trial paths, bookkeeping hashes, official labels, and future observations remain outside it. Repository-relative code and test paths stay visible.

CPU seed	Accuracy	Brier	Hours
0	.781	.175	2.05
1	.719	.191	1.91
2	.813	.187	1.93

Table 4. Final epoch-three development results. Hours are elapsed time per process; the runs overlap in wall time.

B CPU measurements

The LoRA runs use rank 16, learning rate 10^{-4}, effective batch 32, and three epochs. Every seed reaches its lowest development log loss at the common final epoch. Inputs have a 4,096-token cap.

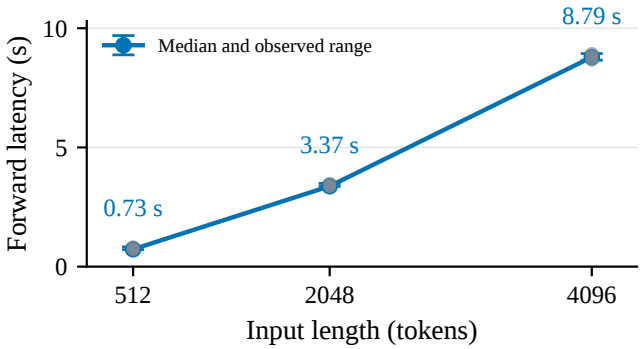


Figure 5. CPU latency. Eight threads, FP32, batch one. Medians and ranges use three timed repetitions after warmup. Tokenization and loading are excluded.

The hosts use Intel Xeon Platinum 8350C processors. Profiling uses fixed-length synthetic token sequences with no generation or persistent KV cache. Cumulative peak RSS is 4.13 GiB through inference. A subsequent 1,024-token training-step probe takes 6.976 seconds and raises that cumulative peak to 4.97 GiB; it is not a model-quality result.

Development intervals resample eight tasks 5,000 times. LoRA metrics are averaged across seeds before task resampling, without probability ensembling. Intervals condition on the trained models; individual seed ticks show the observed run variation.