WHERE DOES THE FAULT LIE? TRACEABLE REPAIR FOR CODING AGENTS BEYOND PASS/FAIL

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic coding systems are increasingly evaluated beyond terminal pass/fail outcomes, yet a passing patch can still be broad, weakly localized, or produced by wasteful repair loops. Across eight models on three external CLI benchmarks, native outcome correlates only weakly with focus, scope, and stability computed from action logs. We formalize this mismatch as an outcome-only traceability gap: the latent repair-state uncertainty left unresolved by the terminal label. To measure this gap directly, we introduce TRACEBENCH-CLI, a 126-task benchmark built from hard Codeforces-derived problems and packaged as stage-gated CLI repair environments. TRACEBENCH-CLI records fault attribution, edit locality, stage progress, wasted turns, and explicit diagnostic behavior. Building on this measurement, we propose ARC (Accountable Repair Control), a training-free fixed-interface controller that reweights frozen-model actions toward localized, diagnostic, and non-cyclic repair decisions. Across five base models on TRACEBENCH-CLI, fixed ARC lifts the Qwen3-32B solve rate from 21.4% to 34.1% and, on the GPT-5.4 models, cuts wasted turns by up to $7.6\times$ and increases explicit diagnosis by up to $3.3\times$. With the same number of model calls per turn, ARC keeps lower waste and more diagnosis than best-of-N and heuristic reranking; for GPT-5.4, its API cost per completed run is $0.74\times$ and $0.65\times$ theirs. On external CLI benchmarks without annotated fault regions, ARC improves focus, scope, and stability without retraining. Code: https://anonymous.4open.science/r/Tracebench-cli/.

1 INTRODUCTION

Coding agents inspect files, edit code, and check results over multiple turns (Yang et al., 2024; Jimenez et al., 2024; Yang et al., 2023; Merrill et al., 2026). Evaluating these steps explains how they reach a result (He et al., 2025; Wang et al., 2025; Liu et al., 2026; Kim et al., 2026). Even successful repairs often contain large, poorly localized patches (Xia et al., 2024). In Figure 1, a focused fix and a broad rewrite both pass, although the latter includes changes unsupported by observed failures. We ask: *Did the agent locate the fault, support its edits with evidence, and avoid repeated work?* We call a repair *traceable* when its trajectory answers these questions.

Command line interface (CLI) and repository benchmarks record parts of this process: SWE-bench reports outcomes (Jimenez et al., 2024), TerminalBench records commands (Merrill et al., 2026), and AgencyBench reports process indicators (Li et al., 2026c). Attribution and edit locality require known fault locations.

We describe a repair by what remains broken, where the fault is, and how much unnecessary work has accumulated. The uncertainty about this state after observing pass/fail is the *traceability gap*. We analyze what observations and recorded actions reveal about this state. TRACEBENCH-CLI makes repair behavior measurable through 126 tasks with known fault regions, the code regions containing injected faults. Its `status` and `test` commands report progress and test the current stage, and `blame` records the fault location that the agent names. *Diff precision* measures edit locality as the fraction of changed lines within the recorded fault regions.

These repair questions guide ARC, which scores a frozen model's proposed actions for diagnosis, revision, and verification. It favors diagnosis while a stage remains unresolved and testing right after edits, and penalizes repeated work and edits outside likely fault locations. It keeps the model's

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.

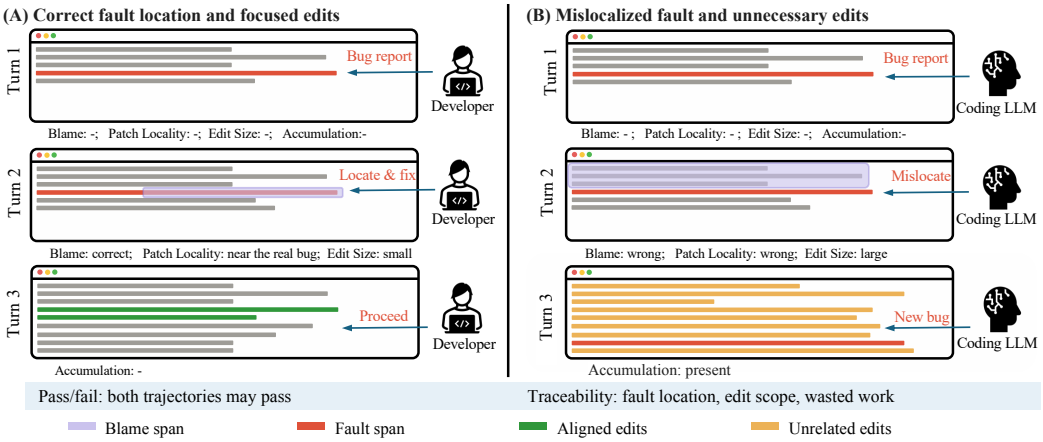


Figure 1: **Passing outcomes can hide differences in repair quality.** Both trajectories may pass, despite differences in fault attribution, edit locality, and unnecessary work.

preferred action unless a fixed rule calls for intervention. DIRECT is a vanilla agent loop that sees command history and test outputs; HARNESS adds the diagnostic tool.

Contributions.

- We formalize what pass/fail leaves unknown about a repair (its stage, fault, and drift) and split it into what observations reveal, what recorded actions add, and what remains (Section 3).
- We introduce TRACEBENCH-CLI, a CLI repair benchmark with known fault regions and staged tests for measuring fault attribution, edit locality, and repair progress (Section 4).
- We propose ARC, a training-free controller that scores a frozen model's candidate actions for diagnosis, verification, edit spread, and repetition, and changes the model's choice only when one of three fixed triggers fires (Section 5).
- With fixed settings across five base models, ARC raises the Qwen3-32B solve rate from 21.4% to 34.1% ($p < 0.01$) and, across the GPT-5.4 family, cuts wasted turns by up to $7.6\times$ (Section 6).

2 RELATED WORK

Code evaluation with tests and execution feedback. HumanEval and MBPP (Chen et al., 2021; Austin et al., 2021) evaluate generated programs through test outcomes. SWE-agent (Yang et al., 2024), SWE-bench (Jimenez et al., 2024), InterCode (Yang et al., 2023), and TerminalBench (Merrill et al., 2026) support interactive execution and primarily report task success. TRACEBENCH-CLI tracks repairs with known fault regions, staged tests, and diagnostic signals.

Self-correction, reflection, and search. Methods such as Self-Debug (Chen et al., 2024), Reflexion (Shinn et al., 2023), Self-Refine (Madaan et al., 2023), ReAct (Yao et al., 2023b), Tree-of-Thoughts (Yao et al., 2023a), and chain-of-thought prompting (Wei et al., 2022) encourage models to inspect failures, reason through alternatives, or retry. Tree search methods such as LATS (Zhou et al., 2024) and SWE-Search (Antoniades et al., 2025) score multiple execution branches with value estimates or reflections. Localization agents such as AutoCodeRover (Zhang et al., 2024) add code search based on syntax trees and spectrum-based fault localization. ARC scores proposed diagnose, revise, and verify actions at one state and executes one after checking its triggers. It works with a frozen agent and its tools on a single execution path.

Evaluating repair trajectories. Recent benchmarks examine agent trajectories. CodeFlowBench (Wang et al., 2025), ContextBench (Li et al., 2026b), RACE-bench (Liu et al., 2026), TRAJEVAL (Kim et al., 2026), CodeTracer (Li et al., 2026a), TRAJECT-Bench (He et al., 2025), and Agent-ProcessBench (Fan et al., 2026) capture context retrieval, reasoning steps, edit targeting, or tool use. Process reward models (Zheng et al., 2025; Choudhury, 2025) similarly supervise individual steps. TRAJEVAL also compares repair steps offline with reference patches. TRACEBENCH-CLI

adds validated fault regions, staged tests, and explicit fault attributions. ARC uses a fixed utility at decision time without a trained process reward model.

Inference-time control of frozen agents. SWE-PRM (Gandhi et al., 2025) redirects a frozen SWE-agent with natural language feedback based on a manually specified list of inefficiencies. ARC scores a finite set of candidate actions and changes the base choice when an intervention trigger fires. Its exponential weighting follows KL-regularized control (Mudgal et al., 2024; Korbak et al., 2022).

3 The Traceability Gap

Passing says little about process quality. Across eight models on three external CLI benchmarks, task scores correlate weakly with the *traceability proxy*, the mean of focus, scope, and stability computed from action logs (Table A.1): $R^2 < 0.04$ and Spearman $\rho < 0.20$ (Figure 2). Focus measures whether diagnostic evidence precedes action on the same resource, scope measures concentration across touched files, and stability measures the fraction of turns that are not wasted.

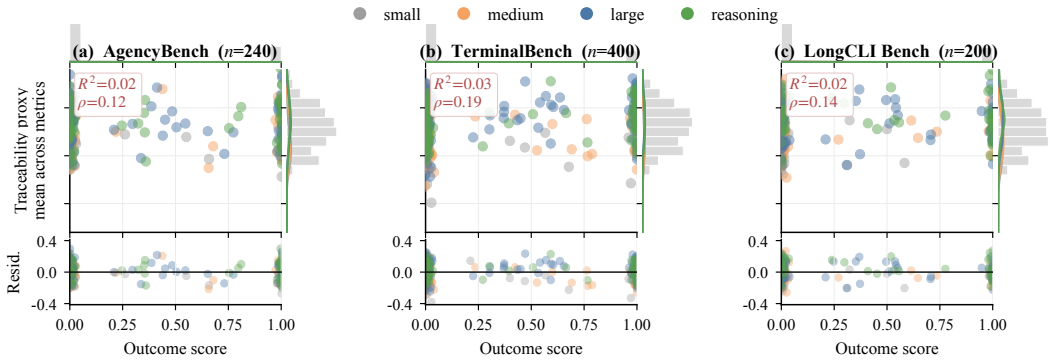


Figure 2: **Outcome and process scores are weakly correlated.** Each point is a task run by one of eight models, colored by model tier (Table D.1). Native task scores are plotted against mean focus, scope, and stability from action logs (Appendix A.5). All three benchmarks have $R^2 < 0.04$ and Spearman $\rho < 0.20$.

What the trajectory shows and what we want to know. Let $e \in \{\mathrm{D}, \mathrm{H}\}$ denote Direct or Harness, and T the number of turns. At turn t, the agent receives observation o_t^e, chooses mode $m_t \in \{\text{diagnose}, \text{revise}, \text{verify}\}$, following the localize, repair, and validate structure of agentic code repair (Xia et al., 2024; Gandhi et al., 2025; Yang et al., 2024), and executes action a_t. Its trajectory is

$$\tau^e = (o_1^e, m_1, a_1, \ldots, o_T^e, m_T, a_T).$$

We write $O^e = o_{1:T}^e$ for observations and $U = (m_{1:T}, a_{1:T})$ for agent decisions, so $\tau^e = (O^e, U)$.

Let s_t record the stage (what remains broken), f_t the fault location, and c_t the accumulated unnecessary work, called drift. The repair state at turn t is

$$z_t = (s_t, f_t, c_t).$$

Table 1 maps these three questions to metrics. Write $Z = z_{1:T}$ for the state sequence and $Y(\tau)$ for the final pass/fail result. Let $\bar{Q}_e$ be the joint distribution of trajectories, states, and outcomes (τ^e, Z, Y) (Appendix A.1); H denotes entropy and I mutual information.

Table 1: Three audit questions and the TraceBench-CLI metrics that answer them (Section 4).

Component	Audit question	What TraceBench-CLI reports
Stage s_t	What remains broken?	stage progress
Fault f_t	Where is the active fault?	`blame` locations, diff precision, minimal-span ratio
Drift c_t	Has unnecessary work accumulated?	wasted turns, exact repeats

How much the pass/fail label leaves unresolved. The final pass/fail result leaves the repair state uncertain. We call this uncertainty the *outcome-only traceability gap*:

$$\mathcal{G}_e^{\text{out}} := H_{\bar{Q}_e}(Z \mid Y). \tag{1}$$

After reading the full trajectory, uncertainty is $\mathcal{R}_e := H_{\bar{Q}_e}(Z \mid \tau^e, Y)$. The decomposition separates the contributions of observations and recorded actions:

Theorem 3.1 (Traceability-gap decomposition)**.** *For any execution setting e,*

$$\mathcal{G}_e^{\text{out}} = \underbrace{I_{\bar{Q}_e}(Z; O^e \mid Y)}_{\mathcal{C}_e^{\text{obs}}\ \textit{(observation closure)}} + \underbrace{I_{\bar{Q}_e}(Z; U \mid O^e, Y)}_{\mathcal{C}_e^{\text{act}}\ \textit{(action closure)}} + \underbrace{H_{\bar{Q}_e}(Z \mid O^e, U, Y)}_{\mathcal{R}_e\ \textit{(residual)}}. \tag{2}$$

The proof expands $H(Z \mid Y)$ using $\tau^e = (O^e, U)$ and the chain rule (Appendix A.2). On TRACEBENCH-CLI, held-out predictors of the repair state (Appendix B.6) show that observations account for nearly all of the reduction for stage and fault, while recorded actions mainly reduce uncertainty about drift: for drift, observations remove 0.70 of the 0.93 outcome-only log loss and actions a further 0.16, and for stage and fault actions remove at most 0.01 (Table B.1).

Interface access and action selection. When the DIRECT observations are a post-processing of the HARNESS observations given the outcome, HARNESS leaves no more uncertainty about the repair state (Proposition A.2, Appendix A.3).

4 TRACEBENCH-CLI

TRACEBENCH-CLI contains 126 CLI repair tasks. Each task runs in a Docker container with a buggy program and hidden tests grouped into stages that must be passed in order. Every task begins with one injected environment fault, so Stage 0 requires locating a build or import error; each later stage runs hidden tests for one subproblem of the reference solution, in dependency order, ending with the complete program. Success requires passing all stages within budget. The diagnostic tool provides three `tracebenchctl` commands: `status` reports stage progress, `test` runs the current stage's hidden tests without revealing them, and `blame` writes the agent's current fault hypothesis, a file and line range with a note, into the trajectory. Section 6.5 walks through one task.

Tasks derive from hard Codeforces problems and follow the TerminalBench format; problems at this level (mean rating 2677) have executable references, so fault regions and stage tests can be validated by replay. We validated reference programs from models and humans by execution, then built staged tests from their executable function specifications. We injected environment faults and changes to the abstract syntax tree (AST), recording fault regions and patches that restore the reference code. Replay verifies the faults, recorded regions, and stage checks. The evaluator uses the regions; the agent and controller see only observable evidence, and the hidden test suite determines correctness.

Tasks vary in stage count (3 to 7), number of injected logic bugs (1 to 4), and environment fault template: E1, a generated source file written to the wrong path, or E2, a fatal typo in a package initializer (Appendices C.1–C.7).

Diff precision measures edit location: the fraction of changed lines within the fault region. Minimal span ratio compares edit size with region size: the region's length divided by edited lines, capped at 1. Wasted turns count retests without a new edit or diagnosis, plus empty or truncated turns. Exact repeats count identical commands or edits repeated within a stage. Explicit diagnosis counts `blame` calls; stage progress averages the fraction of stages passed per task. With $\text{Waste}_{\max} = \text{Blame}_{\max} = 20$, composite process quality is $\frac{1}{4}[\text{Diff} + \text{MinSpan} + (1 - \text{Waste}/\text{Waste}_{\max}) + \text{Blame}/\text{Blame}_{\max}]$. Components are in Appendix A.5.

5 ACCOUNTABLE REPAIR CONTROL

Choosing the next action. ARC reads test outputs and action history, updates likely fault locations, and scores the model's proposed actions. It retains the base choice unless a trigger fires, then selects the allowed candidate with the highest adjusted score. Because recorded actions mainly resolve drift (Section 3), ARC penalizes actions that add drift and records explicit fault attributions with `blame`. Subscripts θ, ϕ, and ψ mark the frozen model, fixed belief rules, and fixed scoring rules.

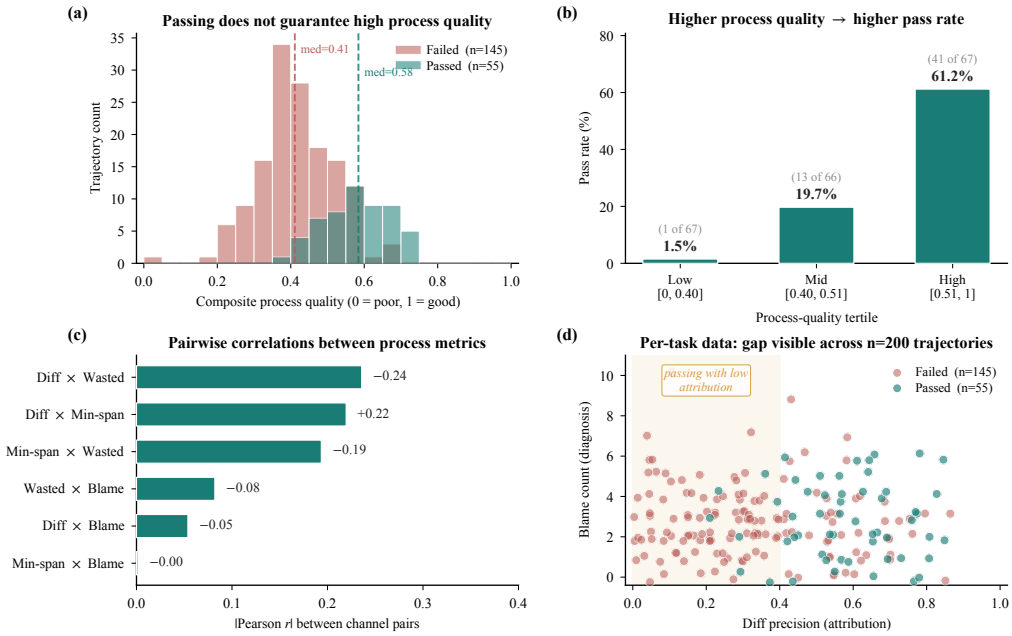


Figure 3: **Process metrics from TRACEBENCH-CLI trajectory logs.** Four views of $n = 200$ trajectories: process quality by outcome, pass rate by process tertile, correlations among metrics, and the relation between diff precision and diagnosis.

Table 2: **Candidate scoring terms**, weights, and evidence used at inference.

Term	Weight	Plain meaning	Deployed proxy (evidence read)	Evaluation metric
Fault information, $\widehat{\Delta H}_F$	α=1.0	Fault diagnosis	Largest stage risk, if diagnostic	Blame
Stage information, $\widehat{\Delta H}_S$	η=0.5	Stage verification	Test after a recent edit, else zero	Waste
Spread, Φ_{spr}	γ_{spr}=0.7	Editing outside likely fault	Edit scope and likely faulty files	Diff precision
Loop, Φ_{loop}	γ_{loop}=1.0	Repeated stalled actions	Recent repeated or empty actions	Waste, repeats

From whole runs to current candidates. At turn t, the pair $u_t = (m_t, a_t)$ records the mode and action, and $h_t^e = (o_1^e, u_1, \ldots, o_{t-1}^e, u_{t-1}, o_t^e)$ is the history. Over whole runs, the ideal controller reweights the frozen model's trajectory distribution p_θ^e by $\exp\{\beta\, \mathcal{Q}_e(\tau)\}$, where $\mathcal{Q}_e$ adds to the outcome score rewards for diagnosis ($\mathcal{U}_{\text{diag}}$) and verification ($\mathcal{U}_{\text{ver}}$; per turn, $\widehat{\Delta H}_F$ and $\widehat{\Delta H}_S$ below) and penalties for spread and loops; this is the unique minimizer of $\beta^{-1}\mathrm{KL}(q \,\|\, p_\theta^e) - \mathbb{E}_q[\mathcal{Q}_e(\tau)]$ (Appendix B.2). Choosing the next action under it would require averaging over future actions and outcomes, so ARC applies the same weighting to the current candidates with a per-turn utility.

Scoring the current candidates. At turn t, ARC receives N candidate actions, $\mathcal{C}_t^e = \{u_i = (m_i, a_i)\}_{i=1}^N$, where a fixed parser assigns each command a_i its mode m_i. Its belief q_ϕ is a set of risk scores, one per stage, per file, and for recent action patterns: a score starts at zero, is updated as $r \leftarrow \lambda_{\text{belief}} r + 1$ ($\lambda_{\text{belief}} = 0.85$) each time a failing test or error message names that stage or file, and is reset to zero when a stage passes (Table B.2). The bonuses $\widehat{\Delta H}_F$ and $\widehat{\Delta H}_S$ score diagnosis and verification; Φ_{spr} and Φ_{loop} penalize scattered edits and repetition. With the fixed weights and evidence in Table 2, the utility is

$$\mathcal{U}_\psi^e(u; h_t^e) = \alpha\, \widehat{\Delta H}_F(u, h_t^e) + \eta\, \widehat{\Delta H}_S(u, h_t^e) - \gamma_{\text{spr}} \Phi_{\text{spr}}(u, h_t^e) - \gamma_{\text{loop}} \Phi_{\text{loop}}(u, h_t^e). \tag{3}$$

For each candidate $u \in \mathcal{C}_t^e$, write $U_t(u) := \mathcal{U}_\psi^e(u; h_t^e)$ for its utility and $\hat{\pi}_t(u) := \hat{\pi}_{\theta,t}^e(u \mid h_t^e)$ for its probability under the frozen base model. ARC forms the reweighted candidate distribution

$$\widetilde{\pi}_{\beta,t}(u) \propto \hat{\pi}_t(u) \exp\{\beta U_t(u)\}, \qquad u \in \mathcal{C}_t^e, \tag{4}$$

with normalization over $\mathcal{C}_t^e$. This applies the exponential weighting above to the current candidates (Appendix B.3); expected utility under $\widetilde{\pi}_{\beta,t}$ is non-decreasing in β (Appendix B.4).

When to change the base choice. The base model's preferred action is $u_b := \arg\max_{u \in \mathcal{C}_t^e} \hat{\pi}_t(u)$. Let u_a be the other candidate with the highest utility, provided that $U_t(u_a) > U_t(u_b)$. Write $S_t(u) := \log \hat{\pi}_t(u) + \beta U_t(u)$ for the adjusted score, which equals $\log \widetilde{\pi}_{\beta,t}(u)$ up to a constant; ARC ranks candidates by S_t rather than sampling from $\widetilde{\pi}_{\beta,t}$, and $G_t := S_t(u_a) - S_t(u_b)$ is the score difference. We check the three triggers in order and record each intervention under the first match.

(1) *Hard reject.* u_b reaches loop limit ρ_{loop} or exceeds spread limit ρ_{spr}. Violating candidates are removed.
(2) *Diagnostic trigger.* The largest stage risk exceeds θ_{diag} and the last k_{diag} turns contain no diagnostic action. The diagnostic candidate with the highest adjusted score is executed; if none exists, ARC records a `blame` on the file with the highest risk score (in DIRECT, it reads that file instead).
(3) *Minimum-utility trigger.* u_a exists and $G_t > \theta_{\min}$. Then u_b is removed. Since $\theta_{\min} < 0$, this happens whenever u_a's adjusted score is within $|\theta_{\min}|$ of u_b's, even if u_b still scores higher.

Without a trigger, ARC executes u_b. After trigger (1) or (3), it selects the candidate with the highest adjusted score S_t among the remaining candidates $\mathcal{R}_t$.

With logit access, token probabilities give $\hat{\pi}_t$ directly. With API access, ARC samples $N = 3$ candidates at temperature 0.7 and an auxiliary pool of N generations at temperature 1; $\hat{\pi}_t$ is each candidate's frequency in this pool (Appendix D.4). In DIRECT, the model runs in a vanilla coding-agent loop without `tracebenchctl`: it sees only the output of tests it writes and runs itself, stage risk comes from those outputs, diagnostic actions are its own inspection and test commands, and it makes no `blame` calls. The model stays frozen; hyperparameters are selected on 16 development tasks disjoint from the 126 evaluation tasks and reused throughout (belief updates, algorithm, and checks in Appendices B.7–B.9).

6 EXPERIMENTS

6.1 SETUP

Our 2×2 design combines DIRECT (vanilla interaction with the task) or HARNESS (with `tracebenchctl`) with ARC off or on; every paired comparison keeps the model and interface fixed. Qwen3-32B in DIRECT (logits available) and gpt-4o-mini in HARNESS are the primary pairs, and the gpt-4o-mini pair also serves the trigger analysis and utility ablations (Section 6.3, Figure 4). Qwen3-32B, GPT-5.4, GPT-5.4-mini, and GPT-5.4-nano are run as further HARNESS pairs with the same hyperparameters (Figure 5(a)), and a frozen Qwen2.5-Coder-14B endpoint serves the selection-rule comparison at matched compute (Table 3(a)). Each run has a budget of 20 turns and 80 tool calls, with 32K/4K input/output tokens per turn; ARC uses $\beta = 0.5$, $\theta_{\min} = -0.5$, $\theta_{\text{diag}} = 0.5$, $k_{\text{diag}} = 3$, $\rho_{\text{loop}} = 1.0$, and $\rho_{\text{spr}} = 0.6$ (Table D.2; compute controls in Appendix D.13); scaling any weight by 0.5–$1.5\times$ preserves the top choice on 99.2% of logged turns (Appendix D.5).

McNemar's paired test compares solve rates; percentile bootstrap gives confidence intervals for process metrics. AgencyBench (Li et al., 2026c), TerminalBench (Merrill et al., 2026), and LongCLI Bench (Feng et al., 2026) provide repair tasks without annotated fault regions; on them we measure focus, scope, and stability (Section 3) for GPT-5.4, gpt-4o-mini, Llama-3.3-70B, and Claude Opus 4.6 (Figure 5(b,c); Appendices D.1–D.7). There, ARC treats file inspection, `grep`, and traceback reads as diagnostic actions and builds its file belief from failed-command output (Appendix D.1).

6.2 DIAGNOSING THE TRACEABILITY GAP ON TRACEBENCH-CLI

Figure 3 combines $n = 200$ HARNESS trajectories on TRACEBENCH-CLI, the same 50 tasks run by gpt-4o-mini, GPT-5.4, GLM-5, and o4-mini, using their known fault regions and `blame` logs; the paired ARC comparisons use all 126 tasks.

Passing and failed runs overlap in process quality: their medians are 0.58 and 0.41, and the two distributions overlap across the middle of the range (a). Pass rate rises from 1.5% in the lowest third of composite process quality to 19.7% in the middle and 61.2% in the highest third, where 26 of 67 runs still fail (b). The metrics are weakly correlated with each other, largest $|r| = 0.24$ (c), and both

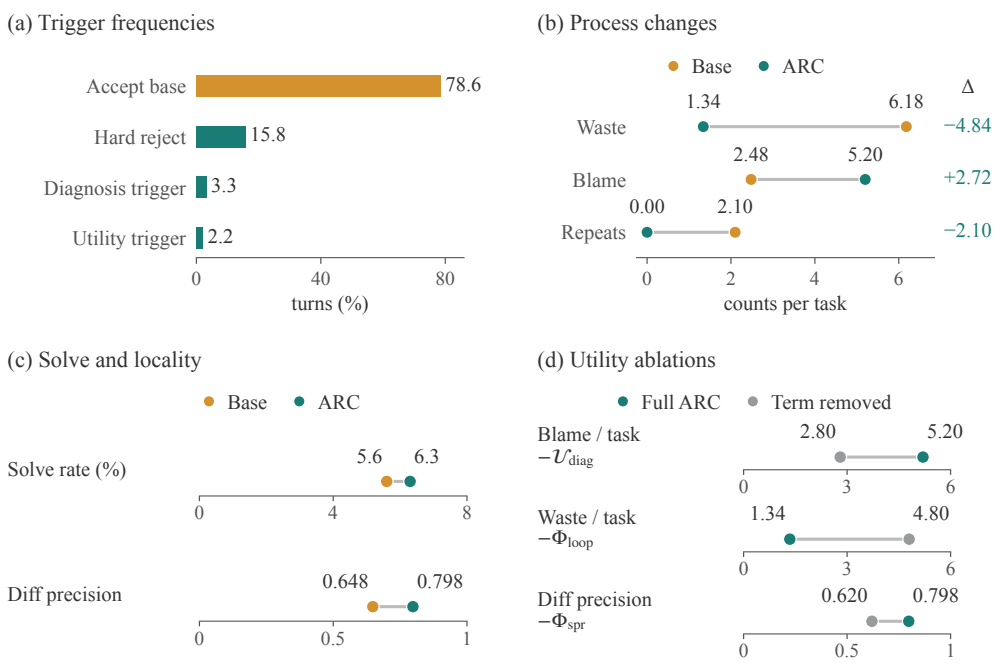


Figure 4: **ARC shifts action selection toward traceable repair.** The gpt-4o-mini harness pair: (a) trigger frequencies; (b) means and changes (ARC minus Base); (c) solve rate and diff precision; (d) utility ablations. Panels (b,c) cover 126 tasks; rows in (c,d) use separate scales.

outcome groups include trajectories with Diff < 0.4 (d). Unlike the log proxies in Figure 2, these direct metrics are informative about the outcome; the label alone still leaves the repair state open, as the overlap in (a) and the failures in the highest third of (b) show. Table C.5 audits the passing patches for required edits outside the recorded regions and replays all 252 final workspaces, with 3 regressions per condition and reference agreement on $993/1000$ (Base) and $1000/1000$ (ARC) fresh inputs.

6.3 ARC SHIFTS ACTION SELECTION TOWARD TRACEABLE REPAIR

ARC retains the base action on 78.6% of turns in the gpt-4o-mini HARNESS pair. The remaining turns involve hard rejection (15.8%), θ_{diag} (3.3%), or $\theta_{\min}$ (2.2%) (Figure 4(a)). Per task, it has 4.84 fewer wasted turns, 2.10 fewer repeated commands, and 2.72 more `blame` calls per task (2.08 issued by the model and 3.12 recorded by ARC when it overrides; b, Table D.5); diff precision rises from 0.648 to 0.798 and solve rate from 5.6% to 6.3% (c). Removing $\mathcal{U}_{\text{diag}}$ reduces `blame` calls ($5.20 \to 2.80$), removing Φ_{loop} raises wasted turns ($1.34 \to 4.80$), and removing Φ_{spr} lowers diff precision ($0.80 \to 0.62$) (d).

The gain persists with the same number of model calls per turn. The four selection rules use the frozen Qwen2.5-Coder-14B endpoint in HARNESS, 126 tasks, a limit of 20 turns, and three proposal plus three auxiliary generations per executed turn. Best-of-N executes the candidate with the highest estimated base probability, and heuristic rerank applies hand-coded rules that favor diagnosis first, local edits, and no repeats. ARC solves $10/126$ tasks against $3/126$ for the base (exact two-sided McNemar $p = 0.016$), with higher stage progress and fewer wasted turns (Table 3(a)). Diff precision among trajectories with edits rises from 0.035 to 0.167. On 400 replayed states with shared candidates, ARC chooses more correct diagnoses and useful verifications than the base (Table D.10).

In the OpenAI comparison (Table D.8), with up to 10 tasks per model and rule, ARC runs completed without API errors have fewer wasted turns and more diagnostic calls than best-of-N and heuristic rerank. For GPT-5.4, its API cost per completed run is $0.74\times$ the best-of-N cost and $0.65\times$ the heuristic rerank cost.

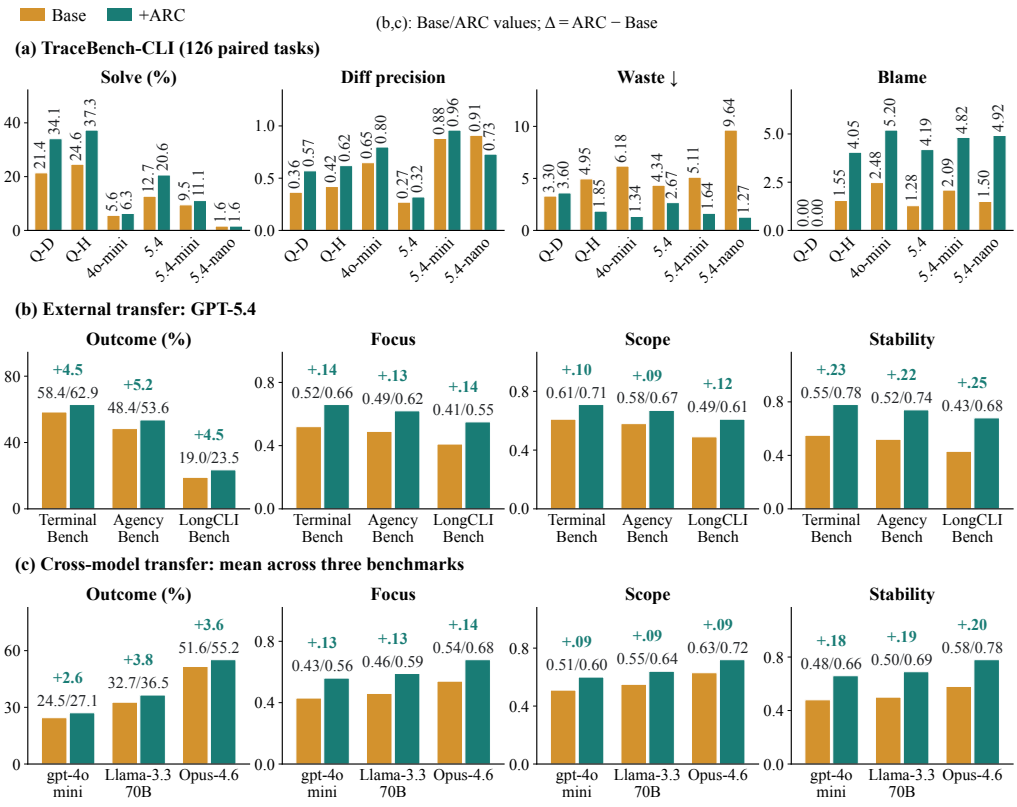


Figure 5: **ARC paired results across models and benchmarks.** Q-D/Q-H: Qwen3-32B in DIRECT/HARNESS; other labels in (a) denote GPT models. Panels show TRACEBENCH-CLI, GPT-5.4 external transfer, and cross-model averages over three benchmarks. Table D.11 lists all values.

Table 3: **Matched sampling and independent repair preferences.** (a) Qwen2.5-Coder-14B, 126 tasks, six generations per turn; Stage and Waste are means per task. (b) Overall quality judgments by three non-author reviewers on 30 external-benchmark pairs matched on task, model, budget, and outcome. A condition wins with at least two votes; other pairs are ties.

(a) Matched sampling

Rule	Solve	Stage	Waste
Base (first proposal)	3/126	0.064	8.56
Best-of-N	3/126	0.056	8.90
Heuristic rerank	2/126	0.070	5.71
ARC	**10/126**	**0.121**	**0.33**

(b) Independent preferences

Preference	Judgments	Pairs
ARC	64/90	22/30
Base	11/90	3/30
Tie	15/90	5/30

6.4 ARC PERFORMANCE ACROSS BENCHMARKS

With fixed controller settings, Qwen3-32B solve rate on TRACEBENCH-CLI (Figure 5(a)) rises from 21.4% to 34.1% ($p < 0.01$, McNemar). Across GPT-5.4 models, wasted turns fall by 1.6–7.6× and `blame` calls rise by 2.3–3.3×; in the Qwen3-32B DIRECT pair, waste and repeats are 3.30 vs 3.60 and 0.33 vs 0.36 per task (Table D.11). The Qwen3-32B HARNESS pair rises from 24.6% to 37.3%, with waste 4.95→1.85 and `blame` 1.55→4.05 per task. For GPT-5.4 on the three external benchmarks (b), focus, scope, and stability improve by 0.09–0.25 (16–58% relative) across the nine benchmark and metric combinations, and pass rates rise by 4.5 to 5.2 percentage points. gpt-4o-mini, Llama-3.3-70B, and Claude Opus 4.6 show comparable gains (c).

Independent reviewers compared 30 outcome-matched pairs, ten from each external benchmark, and preferred ARC on 22 (Table 3(b)). Reviewer agreement is Fleiss' $\kappa = 0.54$, and a sign test over pairs gives $p = 1.6 \times 10^{-4}$. Figure 6 shows gains of 0.6 to 0.9 points on all five ratings. On a separate set of 12 naturally occurring bugs, the number solved rises from 3 to 4 and wasted turns fall from 5.08 to 1.67 (confidence intervals and protocols in Appendices D.6 and D.16).

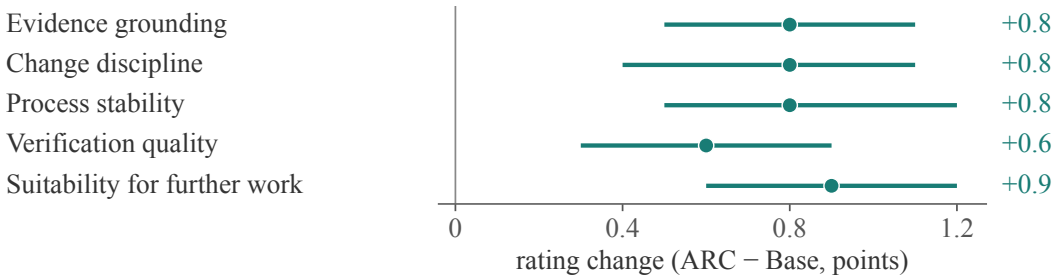


Figure 6: **Independent repair ratings.** Mean changes on a five point scale from three non-author reviewers of 30 matched pairs; bars show 95% pair bootstrap confidence intervals.

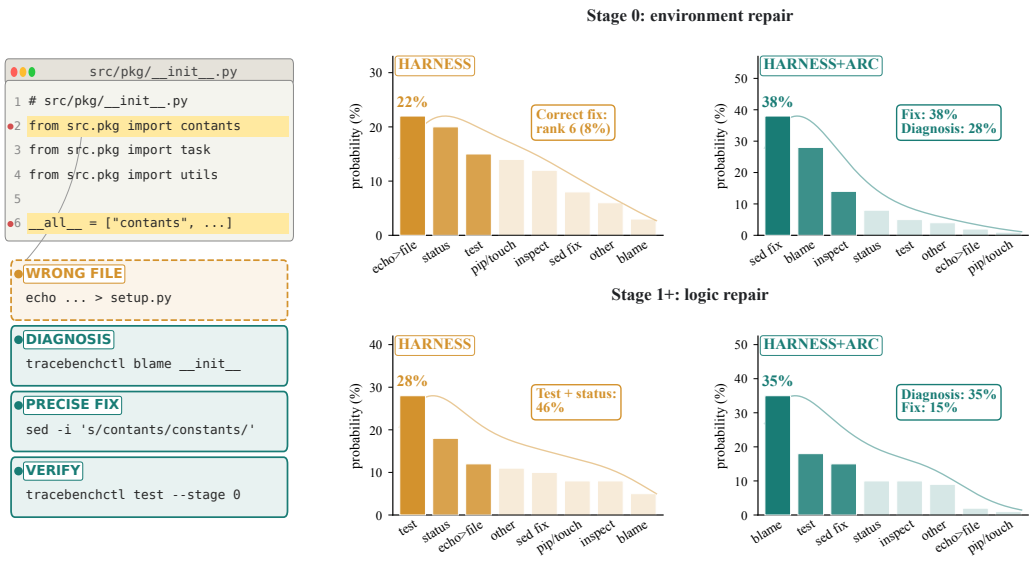


Figure 7: **Action probabilities in the case study.** Candidate action distributions under HARNESS and HARNESS+ARC, pooled over the turns in Stage 0 and in Stage 1+ of the two runs in Appendix E.1 (categories from the fixed action parser). Darker bars mark the top three actions.

6.5 CASE STUDY

Figure 7 compares controllers on the same task (`tb_hard_00045`), model (gpt-4o-mini), interface (HARNESS), and budget. A typo in a package initializer causes the Stage 0 failure. The base model favors editing an unrelated setup file (22%); the correct one-line repair ranks sixth (8%).

HARNESS+ARC moves the correct repair to rank one (38%) and a diagnostic action to rank two (28%). At later stages, HARNESS favors repeated `test` and `status` calls, while HARNESS+ARC favors diagnose, revise, and verify actions. Additional cases are in Appendices E.1–E.4.

7 CONCLUSION AND LIMITATIONS

TRACEBENCH-CLI measures what pass/fail leaves unobserved in a repair: fault attribution, edit locality, stage progress, and repeated work. ARC acts on the action channel, recording explicit fault attributions and suppressing actions that add drift; solve rate, the matched-compute comparison, and the independent ratings measure its effect without the process metrics that its utility terms target. The same measurements apply to any agent's trajectory, with or without ARC. **Limitations.** Faults are injected into single-program Codeforces solutions; the 12 naturally occurring multi-file bugs in Section 6.4 are a first check beyond them. On external benchmarks, focus, scope, and stability are log-based proxies without fault annotations, checked only by the independent ratings in Table 3(b).

AI USE STATEMENT

LLMs assisted with reference program generation, data processing and filtering, manuscript review and editing, and the interpretation and presentation of experimental results and figures.

REFERENCES

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. SWE-search: Enhancing software agents with Monte Carlo tree search and iterative refinement. In *International Conference on Learning Representations*, 2025.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, 2024.

Sanjiban Choudhury. Process reward models for llm agents: Practical framework and directions. *arXiv preprint arXiv:2502.10325*, 2025.

Shengda Fan, Xuyan Ye, Yupeng Huo, Zhi-Yuan Chen, Yiju Guo, Shenzhi Yang, Wenkai Yang, Shuqi Ye, Jingwen Chen, Haotian Chen, et al. AgentProcessBench: Diagnosing step-level process quality in tool-using agents. *arXiv preprint arXiv:2603.14465*, 2026.

Yukang Feng, Jianwen Sun, Zelai Yang, Jiaxin Ai, Chuanhao Li, Zizhen Li, Fanrui Zhang, Kang He, Rui Ma, Jifan Lin, et al. Longcli-bench: A preliminary benchmark and study for long-horizon agentic programming in command-line interfaces. *arXiv preprint arXiv:2602.14337*, 2026.

Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. When agents go astray: Course-correcting SWE agents with PRMs. In *NeurIPS 2025 Workshop on Scaling Environments for Agents*, 2025.

Pengfei He, Zhenwei Dai, Bing He, Hui Liu, Xianfeng Tang, Hanqing Lu, Juanhui Li, Jiayuan Ding, Subhabrata Mukherjee, Suhang Wang, Yue Xing, Jiliang Tang, and Benoit Dumoulin. Traject-bench: A trajectory-aware benchmark for evaluating agentic tool use. *arXiv preprint arXiv:2510.04550*, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Myeongsoo Kim, Dingmin Wang, Siwei Cui, Farima Farmahinifarahani, Shweta Garg, Baishakhi Ray, Terry Yue Zhuo, Rajdeep Mukherjee, and Varun Kumar. Trajeval: Decomposing code agent trajectories for fine-grained diagnosis. *arXiv preprint arXiv:2603.24631*, 2026.

Tomasz Korbak, Ethan Perez, and Christopher Buckley. RL with KL penalties is better viewed as bayesian inference. In *Findings of the Association for Computational Linguistics: EMNLP 2022*, 2022.

Han Li, Yifan Yao, Letian Zhu, Rili Feng, Hongyi Ye, Jiaming Wang, Yancheng He, Pengyu Zou, Lehan Zhang, Xinping Lei, et al. Codetracer: Towards traceable agent states. *arXiv preprint arXiv:2604.11641*, 2026a.

Han Li, Letian Zhu, Bohan Zhang, Rili Feng, Jiaming Wang, Yue Pan, Earl T. Barr, Federica Sarro, Zhaoyang Chu, and He Ye. Contextbench: A benchmark for context retrieval in coding agents. *arXiv preprint arXiv:2602.05892*, 2026b.

Keyu Li, Junhao Shi, Yang Xiao, Mohan Jiang, Jie Sun, Yunze Wu, Dayuan Fu, Shijie Xia, Xiaojie Cai, Tianze Xu, et al. Agencybench: Benchmarking the frontiers of autonomous agents in 1m-token real-world contexts. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7422–7440, 2026c.

Shuhan Liu, Zhiyi Zhao, Xing Hu, Kui Liu, Xiaohu Yang, and Xin Xia. A benchmark for evaluating repository-level code agents with intermediate reasoning on feature addition task. *arXiv preprint arXiv:2603.26337*, 2026.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Sidharth Mudgal, Jong Lee, Harish Ganapathy, YaGuang Li, Tao Wang, Yanping Huang, Zhifeng Chen, Heng-Tze Cheng, Michael Collins, Trevor Strohman, Jilin Chen, Alex Beutel, and Ahmad Beirami. Controlled decoding from language models. In *International Conference on Machine Learning*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Sizhe Wang, Zhengren Wang, Dongsheng Ma, Yongan Yu, Rui Ling, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. Codeflowbench: A multi-turn, iterative benchmark for complex code generation. *arXiv preprint arXiv:2504.21751*, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

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

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Advances in Neural Information Processing Systems*, 2023.

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

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

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023b.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024.

Congmin Zheng, Jiachen Zhu, Zhuoying Ou, Yuxiang Chen, Kangning Zhang, Rong Shan, Zeyu Zheng, Mengyue Yang, Jianghao Lin, Yong Yu, and Weinan Zhang. A survey of process reward models: From outcome signals to process supervisions for large language models. *arXiv preprint arXiv:2510.08049*, 2025.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. In *International Conference on Machine Learning*, 2024.

APPENDIX CONTENTS

A Traceability-Gap Theory and Proxy Definitions **13**

A.1 Full factorization of the repair process 13

A.2 Proof and operational form of Theorem 3.1 14

A.3 Proof and operational form of Proposition A.2 16

A.4 Practical estimators for the latent variables 17

A.5 Direct metrics and benchmark-agnostic proxies 17

B ARC Derivation and Inference Details **19**

B.1 Fixed-interface factorization used by ARC 19

B.2 Proof of Proposition B.1 19

B.3 From the path posterior to the local ARC policy 20

B.4 Proof of the local tilt identities 21

B.5 Signed-shift identity 22

B.6 Information-theoretic interpretation of the utility terms 22

B.7 Practical realization of the utility terms 23

B.8 ARC inference algorithm 24

B.9 Belief misspecification and safeguards 26

C TRACEBENCH-CLI Construction and Task Details **27**

C.1 Source collection, reference solutions, and CLI construction 27

C.2 Task-level statistics 28

C.3 Stage and injected-bug distribution 28

C.4 Environment fault templates 29

C.5 The fixed 126-task manifest 29

C.6 `tracebenchctl` interface 29

C.7 Released trajectory examples for benchmark construction 29

D Experimental Protocol, Controls, and Supplementary Results **31**

D.1 Benchmarks used in this paper 31

D.2 Models used in this paper 31

D.3 Execution settings and backends 32

D.4 Budgets and decoding 32

D.5 Controller hyperparameters 33

D.6 Statistical methodology 33

D.7 Reproducibility pointers 34

D.8 Cross-benchmark proxy views 35

D.9 Additional direct diagnostics for TRACEBENCH-CLI 35

D.10 Distributional evidence for the joint utility 37

D.11 Action uncertainty across completed models 39
D.12 Full ablation table 41
D.13 Compute-matched control suite 41
D.14 All paired numbers on the 126-task manifest 43
D.15 Strong-model controller check 43
D.16 Per-LLM by benchmark transfer breakdown 44

E Case Studies and Forensic Timelines **46**
E.1 Diagnosis-first repair converts a fail into a pass 46
E.2 Diffuse edits versus systematic diagnosis in failed repairs 46
E.3 Both conditions pass, but trajectory quality differs 46
E.4 Matched forensic timeline 46
E.5 Scope and limitations 47

The appendix gives proofs, benchmark and protocol details, and supplementary results.

A Traceability-Gap Theory and Proxy Definitions

This section defines the repair process, proves the traceability gap decomposition, and compares observation interfaces. It then defines the measured state components and metrics.

A.1 Full factorization of the repair process

The main text denotes the audit distribution by $\bar{Q}_e(\tau^e, Z, Y)$. This subsection defines the process model for the posterior over repair states. The model keeps the workspace state sequence $X = x_{1:T}$ explicit. It uses $Z = z_{1:T}$ to summarize the repair process for the auditor; the agent does not observe this state. Let $u_t = (m_t, a_t)$ and

$$h_t^e := (o_1^e, u_1, \ldots, o_{t-1}^e, u_{t-1}, o_t^e)$$

be the agent-visible history at turn t. Here p_{obs}^e is the observation channel induced by the execution setting, π_θ^e is the frozen agent policy over mode-action pairs, p_{env} is the workspace transition induced by the chosen action, and p_{tr} is the repair-state transition used by the auditor. Here p_0 is the initial joint distribution of workspace and repair state, and Θ collects the quantities specifying the process model. With $\tau^e = (o_1^e, u_1, \ldots, o_T^e, u_T)$, define the full process law

$$P_{\Theta,e}^{\mathrm{proc}}(\tau^e, X, Z, Y) = p_0(x_1, z_1) \prod_{t=1}^{T} \Big[p_{\mathrm{obs}}^e(o_t^e \mid x_t)\, \pi_\theta^e(u_t \mid h_t^e) \Big] \cdot \prod_{t=1}^{T-1} \Big[p_{\mathrm{env}}(x_{t+1} \mid x_t, u_t)\, p_{\mathrm{tr}}(z_{t+1} \mid z_t, x_t, x_{t+1}, u_t, o_{t+1}^e) \Big] \mathbf{1}\{Y = Y(\tau^e)\}. \tag{5}$$

The transition for Z depends on the executed action because edits, tests, and diagnoses can change the stage, fault, and drift. The agent policy itself does not condition on Z; it only receives h_t^e.

The transition product stops at $t = T - 1$ because $X = x_{1:T}$ and $Z = z_{1:T}$ are indexed by decision turns; the effect of the final action u_T enters through the terminal label $Y(\tau^e)$, which is evaluated after u_T executes. Appending a terminal pair (x_{T+1}, z_{T+1}) with the same kernels extends the product to $t = T$ and leaves Theorem 3.1 unchanged, because the theorem uses only the joint law of (Z, O^e, U, Y).

Marginalizing the workspace state gives the process posterior over audit states:

$$P^{\mathrm{proc}}_{\Theta,e}(\tau^e, Z, Y) = \int P^{\mathrm{proc}}_{\Theta,e}(\tau^e, X, Z, Y)\, dX.$$

The model summarizes repair through signals that are observable or verifiable offline: the current stage, files or spans implicated by evidence, edit spread, tests, and diagnostic responses.

For the information-theoretic quantities in Section 3, we evaluate traces under the realized trajectory distribution $P_e(\tau^e)$ for setting e. Let $q^\star_\phi(Z \mid \tau^e)$ be the variational audit posterior chosen to approximate $P^{\mathrm{proc}}_{\Theta,e}(Z \mid \tau^e, Y(\tau^e))$ within a fixed family:

$$q^\star_\phi \in \arg\min_{q_\phi} \mathbb{E}_{\tau^e \sim P_e}\Big[\mathrm{KL}\big(q_\phi(\cdot \mid \tau^e) \,\|\, P^{\mathrm{proc}}_{\Theta,e}(\cdot \mid \tau^e, Y(\tau^e))\big)\Big].$$

The audit distribution is then

$$\bar{Q}_e(\tau^e, Z, Y) = P_e(\tau^e)\, \mathbf{1}\{Y = Y(\tau^e)\}\, q^\star_\phi(Z \mid \tau^e). \tag{6}$$

All entropies and mutual informations in the traceability-gap decomposition are computed with respect to $\bar{Q}_e$, which combines logged trajectories with the posterior over repair states.

Information carried by actions. The action sequence U can carry information about Z under the audit distribution. Actions change the workspace, affect later observations, and determine whether the pass/fail outcome is reached. Thus, after conditioning on O^e and Y, the posterior $\bar{Q}_e(Z \mid O^e, U, Y)$ can differ from $\bar{Q}_e(Z \mid O^e, Y)$. The information revealed by this posterior update is captured by the action closure term $I_{\bar{Q}_e}(Z; U \mid O^e, Y)$ in Theorem 3.1.

A.2 Proof and operational form of Theorem 3.1

Let $Z = z_{1:T}$, $O^e = o^e_{1:T}$, and $U = (m_{1:T}, a_{1:T})$. By construction, $\tau^e = (O^e, U)$. All information-theoretic quantities below are evaluated under $\bar{Q}_e$ in Eq. (6); we omit the subscript when unambiguous.

Audit filtration. The decomposition follows an auditor predicting the repair state from progressively more information. Define the three information sets

$$\mathcal{F}_0 := \sigma(Y), \qquad \mathcal{F}^e_1 := \sigma(Y, O^e), \qquad \mathcal{F}^e_2 := \sigma(Y, O^e, U).$$

The auditor receives the outcome under $\mathcal{F}_0$, adds interface observations under $\mathcal{F}^e_1$, and adds actions under $\mathcal{F}^e_2$. These sigma-fields are nested:

$$\mathcal{F}_0 \subseteq \mathcal{F}^e_1 \subseteq \mathcal{F}^e_2.$$

For any information set $\mathcal{F}$, consider an auditor that reports a predictive distribution $r(\cdot \mid \mathcal{F})$ over the repair state Z and is scored by logarithmic loss:

$$\mathcal{L}(r; \mathcal{F}) := \mathbb{E}_{\bar{Q}_e}\big[-\log r(Z \mid \mathcal{F})\big].$$

The optimal audit loss is

$$\mathcal{L}^\star(\mathcal{F}) := \inf_r \mathcal{L}(r; \mathcal{F}) = H(Z \mid \mathcal{F}),$$

with the infimum attained by the posterior $r^\star(z \mid \mathcal{F}) = \bar{Q}_e(z \mid \mathcal{F})$. Indeed,

$$\mathcal{L}(r; \mathcal{F}) = H(Z \mid \mathcal{F}) + \mathbb{E}_{\mathcal{F}}\Big[\mathrm{KL}\big(\bar{Q}_e(Z \mid \mathcal{F}) \,\|\, r(Z \mid \mathcal{F})\big)\Big],$$

and the KL term is minimized at the true posterior.

Thus the outcome-only traceability gap is the optimal log-loss of an auditor that sees only the pass/fail label:

$$\mathcal{G}^{\mathrm{out}}_e = H(Z \mid Y) = \mathcal{L}^\star(\mathcal{F}_0).$$

Adding observations reduces this optimal audit loss by

$$\mathcal{L}^\star(\mathcal{F}_0) - \mathcal{L}^\star(\mathcal{F}^e_1) = H(Z \mid Y) - H(Z \mid O^e, Y) = I(Z; O^e \mid Y).$$

Adding actions reduces it further by

$$\mathcal{L}^\star(\mathcal{F}_1^e) - \mathcal{L}^\star(\mathcal{F}_2^e) = H(Z \mid O^e, Y) - H(Z \mid O^e, U, Y) = I(Z; U \mid O^e, Y).$$

The remaining optimal audit loss after reading the full trace is

$$\mathcal{L}^\star(\mathcal{F}_2^e) = H(Z \mid O^e, U, Y).$$

Combining the three displayed equations gives

$$H(Z \mid Y) = \underbrace{I(Z; O^e \mid Y)}_{\mathcal{C}_e^{\mathrm{obs}}} + \underbrace{I(Z; U \mid O^e, Y)}_{\mathcal{C}_e^{\mathrm{act}}} + \underbrace{H(Z \mid O^e, U, Y)}_{\mathcal{R}_e},$$

which is Eq. (2).

Posterior-update form. The two closure terms can also be written as expected posterior updates:

$$\mathcal{C}_e^{\mathrm{obs}} = \mathbb{E}_{Y, O^e}\left[\mathrm{KL}\big(\bar{Q}_e(Z \mid Y, O^e) \,\|\, \bar{Q}_e(Z \mid Y)\big)\right],$$

and

$$\mathcal{C}_e^{\mathrm{act}} = \mathbb{E}_{Y, O^e, U}\left[\mathrm{KL}\big(\bar{Q}_e(Z \mid Y, O^e, U) \,\|\, \bar{Q}_e(Z \mid Y, O^e)\big)\right].$$

This form makes the audit meaning explicit. Observation closure is the average posterior update after reading the interface observations. Action closure is the average posterior update after additionally reading which diagnose, revise, and verify actions the agent selected. Because observations are read first, information that observations and actions share is credited to observation closure; action closure is the information that actions add beyond observations.

Equality cases. The equality cases follow from the posterior-update form:

- $\mathcal{C}_e^{\mathrm{obs}} = 0$ iff $\bar{Q}_e(Z \mid Y, O^e) = \bar{Q}_e(Z \mid Y)$ almost surely, equivalently $Z \perp O^e \mid Y$.
- $\mathcal{C}_e^{\mathrm{act}} = 0$ iff $\bar{Q}_e(Z \mid Y, O^e, U) = \bar{Q}_e(Z \mid Y, O^e)$ almost surely, equivalently $Z \perp U \mid (O^e, Y)$.
- $\mathcal{R}_e = 0$ iff $\bar{Q}_e(Z \mid O^e, U, Y)$ is degenerate almost surely, equivalently Z is a deterministic function of (O^e, U, Y).

Since the three terms are successive reductions in optimal log-loss plus the remaining optimal log-loss, they partition the outcome-only audit risk. □

We represent the repair state Z in Theorem 3.1 as a sequence of repair states

$$Z = (S, F, C),$$

where $S = s_{1:T}$ is the stage sequence, $F = f_{1:T}$ is the fault sequence, and $C = c_{1:T}$ is the drift sequence. Applying the chain rule to this ordered tuple gives a coding-agent component form of the outcome-only gap:

$$\mathcal{G}_e^{\mathrm{out}} = H(S \mid Y) + H(F \mid S, Y) + H(C \mid S, F, Y).$$

The first term asks whether the pass/fail label reveals which stage remained unresolved; the second asks whether it reveals where the active fault lay once the stage is known; the third asks whether it reveals drift once stage and fault are known.

The observation closure similarly decomposes as

$$\mathcal{C}_e^{\mathrm{obs}} = I(S; O^e \mid Y) + I(F; O^e \mid S, Y) + I(C; O^e \mid S, F, Y),$$

and the action closure decomposes as

$$\mathcal{C}_e^{\mathrm{act}} = I(S; U \mid O^e, Y) + I(F; U \mid O^e, S, Y) + I(C; U \mid O^e, S, F, Y).$$

Finally, the residual decomposes as

$$\mathcal{R}_e = H(S \mid O^e, U, Y) + H(F \mid O^e, U, S, Y) + H(C \mid O^e, U, S, F, Y).$$

This component form explains why coding-agent evaluation cannot be reduced to a single pass/fail label. A pass/fail outcome may partially identify S while leaving F and C ambiguous: the task passed, but the repair may still have touched the wrong region or introduced unnecessary drift. Conversely, a failed trajectory may contain useful information about S or F even though Y is negative. TRACEBENCH-CLI exposes observable projections of these components: stage progress for S, diff precision and minimal-span ratio for F, and wasted turns, repeats, and edit spread for D.

A.3 Proof and operational form of Proposition A.2

Assumption A.1 (Outcome-compatible observation refinement)**.** There exists a coupled reference joint Q over $(Z, Y, O^{\mathrm{H}}, O^{\mathrm{D}})$ whose marginals agree with the corresponding direct and harness settings. Conditional on Y, the observation is a stochastic post-processing of the harness observation:

$$Q(o^{\mathrm{D}}_{1:T} \mid o^{\mathrm{H}}_{1:T}, Z, Y) = K_Y(o^{\mathrm{D}}_{1:T} \mid o^{\mathrm{H}}_{1:T})$$

for some kernel K_Y independent of Z. Conditional on Y, $Z \to O^{\mathrm{H}} \to O^{\mathrm{D}}$ is a Markov chain.

The assumption concerns interface observability, the information exposed under the two interfaces. Marginal agreement is required only for the (Z, Y) marginal, which both settings share as a reference prior, and for each setting's own observation channel $Q(O^e \mid Z, Y)$; the observation spaces differ, and the trajectory distributions of the two settings need not coincide.

Proposition A.2 (Observation-channel dominance)**.** *Under Assumption A.1 and the coupled reference Q, define $\mathcal{C}^{\mathrm{obs}}_{e,Q} := I_Q(Z; O^e \mid Y)$. Then*

$$\mathcal{C}^{\mathrm{obs}}_{\mathrm{H},Q} \geq \mathcal{C}^{\mathrm{obs}}_{\mathrm{D},Q}, \quad \textit{and equivalently} \quad H_Q(Z \mid O^{\mathrm{H}}, Y) \leq H_Q(Z \mid O^{\mathrm{D}}, Y).$$

We prove a slightly stronger conditional refinement statement. Under Assumption A.1, for each value y with $Q(Y = y) > 0$, the conditional law satisfies

$$Q_y(z, o^{\mathrm{H}}, o^{\mathrm{D}}) = Q_y(z, o^{\mathrm{H}})\, K_y(o^{\mathrm{D}} \mid o^{\mathrm{H}}), \qquad Q_y(\cdot) := Q(\cdot \mid Y = y).$$

Thus O^{D} is a Y-compatible garbling of O^{H} for the purpose of auditing Z: conditional on $Y = y$,

$$Z \to O^{\mathrm{H}} \to O^{\mathrm{D}}$$

is a Markov chain.

Audit-risk dominance. Let $\ell(z, d)$ be any audit loss for predicting the repair state, and allow randomized decision rules. For any auditor δ_{D} that observes only O^{D}, define an auditor $\widetilde{\delta}_{\mathrm{H}}$ that observes O^{H} by first sampling $\widetilde{O}^{\mathrm{D}} \sim K_y(\cdot \mid O^{\mathrm{H}})$ and then applying $\delta_{\mathrm{D}}(\widetilde{O}^{\mathrm{D}})$. By construction,

$$\mathbb{E}_{Q_y}\big[\ell(Z, \widetilde{\delta}_{\mathrm{H}}(O^{\mathrm{H}}))\big] = \mathbb{E}_{Q_y}\big[\ell(Z, \delta_{\mathrm{D}}(O^{\mathrm{D}}))\big].$$

Taking the infimum over all O^{D}-auditors gives

$$\inf_{\delta_{\mathrm{H}}} \mathbb{E}_{Q_y}\big[\ell(Z, \delta_{\mathrm{H}}(O^{\mathrm{H}}))\big] \leq \inf_{\delta_{\mathrm{D}}} \mathbb{E}_{Q_y}\big[\ell(Z, \delta_{\mathrm{D}}(O^{\mathrm{D}}))\big].$$

Therefore, under every outcome stratum $Y = y$, the harness observation is at least as informative as the direct observation for any audit task on Z. The mutual-information statement below is the logarithmic-loss special case of this refinement.

Log-loss specialization. For logarithmic audit loss, an auditor reports a distribution $r(\cdot \mid O, Y = y)$ over Z and receives loss $-\log r(Z \mid O, Y = y)$. The optimal risk is the conditional entropy:

$$\inf_r \mathbb{E}_Q[-\log r(Z \mid O, Y) \mid Y = y] = H_Q(Z \mid O, Y = y).$$

Applying the audit-risk dominance above with $O = O^{\mathrm{H}}$ and $O = O^{\mathrm{D}}$ gives, for every y,

$$H_Q(Z \mid O^{\mathrm{H}}, Y = y) \leq H_Q(Z \mid O^{\mathrm{D}}, Y = y).$$

Averaging over Y yields the residual-uncertainty form:

$$H_Q(Z \mid O^{\mathrm{H}}, Y) \leq H_Q(Z \mid O^{\mathrm{D}}, Y).$$

Observation-closure form. Because the coupled reference Q has a common marginal over (Z, Y), both settings share the same $H_Q(Z \mid Y)$. Hence

$$\begin{aligned} \mathcal{C}^{\mathrm{obs}}_{\mathrm{H},Q} - \mathcal{C}^{\mathrm{obs}}_{\mathrm{D},Q} &= I_Q(Z; O^{\mathrm{H}} \mid Y) - I_Q(Z; O^{\mathrm{D}} \mid Y) \\ &= H_Q(Z \mid O^{\mathrm{D}}, Y) - H_Q(Z \mid O^{\mathrm{H}}, Y) \\ &\geq 0. \end{aligned}$$

This proves

$$\mathcal{C}^{\mathrm{obs}}_{\mathrm{H},Q} \geq \mathcal{C}^{\mathrm{obs}}_{\mathrm{D},Q}.$$

Slack as posterior update. The amount by which the harness observation dominates the direct observation has an exact posterior-update form. Since $Z \perp O^{\mathrm{D}} \mid (O^{\mathrm{H}}, Y)$,

$$H_Q(Z \mid O^{\mathrm{H}}, Y) = H_Q(Z \mid O^{\mathrm{H}}, O^{\mathrm{D}}, Y).$$

Therefore

$$\mathcal{C}^{\mathrm{obs}}_{\mathrm{H},Q} - \mathcal{C}^{\mathrm{obs}}_{\mathrm{D},Q} = I_Q(Z; O^{\mathrm{H}} \mid O^{\mathrm{D}}, Y),$$

and equivalently

$$\mathcal{C}^{\mathrm{obs}}_{\mathrm{H},Q} - \mathcal{C}^{\mathrm{obs}}_{\mathrm{D},Q} = \mathbb{E}_Q\left[\mathrm{KL}\big(Q(Z \mid O^{\mathrm{H}}, Y) \,\|\, Q(Z \mid O^{\mathrm{D}}, Y)\big)\right].$$

Thus the dominance gap is the expected posterior update about the latent repair state obtained from the harness observation after the direct observation is already known.

Equality case. Equality holds iff the posterior update above vanishes:

$$Q(Z \mid O^{\mathrm{H}}, Y) = Q(Z \mid O^{\mathrm{D}}, Y) \quad Q\text{-a.s.}$$

Equivalently,

$$I_Q(Z; O^{\mathrm{H}} \mid O^{\mathrm{D}}, Y) = 0.$$

This does not require O^{H} and O^{D} to be identical observations; it only requires them to be equivalent for auditing Z under the coupled reference. In coding-agent terms, the harness may expose more raw logs or commands, but equality holds precisely when those extra observations do not change the posterior over stage, fault, or drift. This completes the proof. □

A.4 Practical estimators for the latent variables

The controller tracks stage, fault, and drift from observable evidence to guide action selection. The benchmark metrics below describe observable aspects of the repair state.

Stage variable. The stage s_t is tracked using stage status, test results, and whether the previous action advanced the task. On TraceBench-CLI, stage-gated tasks provide this evidence.

Fault variable. Evidence about the fault f_t, the file or code region containing the active bug, comes from failed tests, touched files, and overlap between edits and diagnostic evidence. The controller maintains decaying risk scores over these files or regions (Table B.2).

Drift variable. The drift c_t is tracked through edit spread, repeated failed commands, repeated tests without intervening edits, truncated actions, and empty turns. These are the same observable signals used by the spread and loop penalties.

A.5 Direct metrics and benchmark-agnostic proxies

Table A.1 defines the metrics used in the paper. TraceBench-CLI provides direct metrics through its stage structure, known fault regions, and diagnosis commands. External benchmarks use indicators computed from action logs.

Normalization constants and edge cases. For composite process quality, we fix $\mathrm{Waste}_{\max} = 20$ (the turn budget) and $\mathrm{Blame}_{\max} = 20$ across all evaluation runs; the normalized metric values are clipped to $[0, 1]$. Trajectories with no edits receive $\mathrm{Diff} = 0$ and $\mathrm{MinSpan} = 0$ in the per-trajectory composite; aggregate Diff is averaged over edit-bearing trajectories only, as noted in captions.

For the focus score, $\mathrm{TargetedDiag}(a_t)$ contains diagnostic actions whose referenced file, directory, symbol, traceback frame, or command target overlaps the resource touched by a_t; on TraceBench-CLI overlap is computed at span/file level using the trajectory log, on external benchmarks at file level. The high-impact set H contains edit/write actions and non-trivial tool calls (build/run/test, package install, destructive shell commands, benchmark submit), excluding read-only filesystem queries (`ls`, `pwd`, `cat`).

For the scope score, $H_K / \log K_t$ is the normalized entropy, so scope measures concentration among the K_t touched resources and is invariant to K_t itself: a trajectory that touches many files but

Table A.1: **Metric and proxy definitions.** Family lists metric types (outcome / direct / proxy/composite) and applicable process axes: attribution (where edits land relative to the fault), propagation (how far edits spread beyond it), accumulation (repeated or wasted work), and diagnosis (explicit fault naming).

Metric	Family	Dir.	Definition
Solve rate	outcome	↑	Fraction of tasks whose final stage passes within budget
Stage progress	outcome	↑	Mean fraction of stages passed per task
diff precision	direct, attribution	↑	Changed lines inside the known fault region divided by total changed lines (averaged over trajectories with edits)
Minimal-span ratio	direct, propagation	↑	$\min\big(1,\ \lvert\text{fault region}\rvert / \lvert\text{edited lines}\rvert\big)$
Wasted turns	direct, accumulation	↓	TWE + truncated + empty turns (per-task count)
TWE (test-without-edit)	direct, accumulation	↓	Test or verification action without an intervening edit or diagnostic update
Exact repeats	direct, accumulation	↓	Count of repeated actions with identical normalized command or edit content within the same stage
Blame calls	direct, diagnosis	↑	Count of `tracebenchctl blame` invocations; higher values indicate more frequent explicit diagnosis
Composite process quality	composite, in-domain	↑	$\frac{1}{4}\big[\,\text{Diff} + \text{MinSpan} + (1-\frac{\text{Waste}}{\text{Waste}_{\max}}) + \frac{\text{Blame}}{\text{Blame}_{\max}}\big]$, equally weighted
Focus score	proxy, attribution	↑	$\frac{1}{\lvert H\rvert}\sum_{a_t\in H}\mathbf{1}\{\exists\, s<t,\ t-s\le k_w,\ a_s\in \text{TargetedDiag}(a_t)\}$, with high-impact set H = edits + non-trivial tool calls and window $k_w=3$
Scope score	proxy, propagation	↑	$1-H_K/\log K_t$, where H_K is the entropy of touched-resource frequencies (file or directory granularity) and K_t the number of distinct resources touched; measures how concentrated activity is among touched resources
Stability score	proxy, accumulation	↑	$1-\text{Waste}/T$, where T is total turns spent in the trajectory
Traceability proxy	composite, external	↑	$(\text{focus}+\text{scope}+\text{stability})/3$

concentrates its edits on one scores high, and a two-file trajectory with equal activity scores zero. Singleton trajectories receive scope $= 1$. The count K_t is available from the same logs, and the controller penalizes edit spread through Φ_{spr}.

For stability on external benchmarks, we count empty or truncated actions, exact command repeats, and verification or run commands repeated without intervening edits or new diagnostic observations.

B ARC Derivation and Inference Details

This section develops the trajectory objective and local action rule, then their identities and utility interpretation. The implementation, inference algorithm, and belief checks follow.

B.1 Fixed-interface factorization used by ARC

For ARC, we use the following model of repair states under a fixed interface, motivated by Eq. (5). The workspace state X is implicit, and the mode and action pair is $u_t = (m_t, a_t)$. We suppress the setting superscript on observations and histories; h_0 is the initial history and $p(z_1)$ the initial repair-state distribution. Here K_e is the observation interface available to the agent, π_θ^e is the frozen base mode-action policy, and P_e summarizes the induced workspace and repair-state transition after executing u_t. The joint distribution is

$$p_\theta^e(\tau, Z, Y) = p(z_1) \prod_{t=1}^{T} \Big[K_e(o_t \mid z_t, h_{t-1}) \pi_\theta^e(u_t \mid h_t) \Big] \prod_{t=1}^{T-1} P_e(z_{t+1} \mid z_t, u_t, h_t)\, \mathbf{1}\{Y = Y(\tau)\}.$$

In a fixed-interface intervention, K_e, P_e, the task manifest, hidden tests, fault annotations, and budgets are held fixed. ARC changes only the action kernel used to select u_t, replacing π_θ^e on a finite candidate set by the reweighted policy in Eq. (4).

B.2 Proof of Proposition B.1

For reference, the trajectory-level utility that enters $\mathcal{Q}_e(\tau) = r(Y(\tau)) + \lambda_{\mathrm{tr}} \widehat{\mathcal{T}}_\psi^e(\tau)$ in Section 5 is

$$\begin{aligned} \widehat{\mathcal{T}}_\psi^e(\tau) = \sum_{t=1}^{T} \Big[& \mathcal{U}_{\mathrm{diag}}(u_t, h_t^e) + \mathcal{U}_{\mathrm{ver}}(u_t, h_t^e) \\ & - \gamma_{\mathrm{spr}} \Phi_{\mathrm{spr}}(u_t, h_t^e) - \gamma_{\mathrm{loop}} \Phi_{\mathrm{loop}}(u_t, h_t^e) \Big]. \end{aligned} \tag{7}$$

The terms reward diagnosis and verification while penalizing spread and loops; per turn they match Eq. (3), with $\mathcal{U}_{\mathrm{diag}} = \alpha \widehat{\Delta H}_F$ and $\mathcal{U}_{\mathrm{ver}} = \eta \widehat{\Delta H}_S$.

The ideal controlled distribution maximizes expected repair quality while staying close to the frozen base trajectory law:

$$q_\psi^{e,\star} = \arg\min_{q \ll p_\theta^e} \Big\{ \beta^{-1} \mathrm{KL}(q \,\|\, p_\theta^e) - \mathbb{E}_q[\mathcal{Q}_e(\tau)] \Big\}. \tag{8}$$

Proposition B.1 (Controlled trajectory posterior). *Assume* $\mathcal{Z}_e := \int p_\theta^e(\tau) \exp(\beta \mathcal{Q}_e(\tau))\, d\tau < \infty$. *The unique minimizer of Eq.* (8) *is*

$$q_\psi^{e,\star}(\tau) = \frac{p_\theta^e(\tau) \exp(\beta \mathcal{Q}_e(\tau))}{\mathcal{Z}_e}.$$

Write $p = p_\theta^e$ and $\mathcal{Q} = \mathcal{Q}_e$ for readability. By assumption,

$$0 < \mathcal{Z}_e := \int p(\tau) \exp(\beta \mathcal{Q}(\tau))\, d\tau < \infty.$$

Define the candidate optimum

$$\bar{q}(\tau) := \frac{p(\tau) \exp(\beta \mathcal{Q}(\tau))}{\mathcal{Z}_e}.$$

By construction $\bar{q}(\tau) \geq 0$ for all τ, $\int \bar{q}(\tau)\, d\tau = 1$, and $\bar{q} \ll p$ since $\bar{q}(\tau) > 0$ only where $p(\tau) > 0$.

Rewrite the objective in terms of $\bar{q}$. For any $q \ll p$, the objective in Eq. (8) is

$$J(q) = \beta^{-1} \mathrm{KL}(q \,\|\, p) - \mathbb{E}_q[\mathcal{Q}(\tau)] = \beta^{-1} \int q(\tau) \log \frac{q(\tau)}{p(\tau)}\, d\tau - \int q(\tau) \mathcal{Q}(\tau)\, d\tau.$$

From the definition of $\bar{q}$,

$$\log \bar{q}(\tau) = \log p(\tau) + \beta \mathcal{Q}(\tau) - \log \mathcal{Z}_e,$$

which rearranges to

$$\log \frac{q(\tau)}{p(\tau)} - \beta \mathcal{Q}(\tau) = \log \frac{q(\tau)}{\bar{q}(\tau)} - \log \mathcal{Z}_e.$$

Substituting into $J(q)$,

$$\begin{aligned} J(q) &= \beta^{-1} \int q(\tau) \left[\log \frac{q(\tau)}{p(\tau)} - \beta \mathcal{Q}(\tau) \right] d\tau \\ &= \beta^{-1} \int q(\tau) \left[\log \frac{q(\tau)}{\bar{q}(\tau)} - \log \mathcal{Z}_e \right] d\tau \\ &= \beta^{-1} \, \mathrm{KL}(q \,\|\, \bar{q}) \; - \; \beta^{-1} \log \mathcal{Z}_e. \end{aligned}$$

Minimize. The second term $-\beta^{-1} \log \mathcal{Z}_e$ is independent of q. By Gibbs' inequality, $\mathrm{KL}(q \,\|\, \bar{q}) \geq 0$ with equality iff $q = \bar{q}$ almost everywhere with respect to the dominating measure of p. Since $\beta > 0$, $J(q)$ is uniquely minimized by $q = \bar{q}$. Therefore

$$q_\psi^{e,\star}(\tau) = \frac{p_\theta^e(\tau) \exp(\beta \mathcal{Q}_e(\tau))}{\int p_\theta^e(\tau') \exp(\beta \mathcal{Q}_e(\tau')) \, d\tau'},$$

as claimed. □

Remarks. First, $\beta \to 0$ recovers $\bar{q} \to p$ (no reweighting), and $\beta \to \infty$ concentrates $\bar{q}$ on $\arg\max_\tau \mathcal{Q}(\tau)$ within the support of p (full reward maximization subject to support constraint). Second, $\bar{q}$ is the standard exponential tilt of p by reward $\mathcal{Q}$ at inverse temperature β, identical in form to the solution of token-level KL-regularized RLHF objectives (Korbak et al., 2022; Mudgal et al., 2024); the new content here is the application at the trajectory level with $\mathcal{Q}_e$ as repair quality.

B.3 From the path posterior to the local ARC policy

Proposition B.1 characterizes the optimal distribution over complete trajectories. We now derive the per-turn conditional distribution that ARC approximates locally.

Sequential factorization. Assume the base trajectory law factorizes sequentially:

$$p_\theta^e(\tau) = p_0(h_1^e) \prod_{t=1}^{T} \pi_\theta^e(u_t \mid h_t^e) \, p_{\mathrm{env}}(h_{t+1}^e \mid h_t^e, u_t),$$

where $u_t = (m_t, a_t)$. Let $g_t(u_t, h_t^e)$ be the reward assigned to turn t. In this subsection, suppose the total reward decomposes additively across turns,

$$\widehat{\mathcal{Q}}_e(\tau) = \sum_{t=1}^{T} g_t(u_t, h_t^e),$$

so that the tilted trajectory law is

$$\bar{q}(\tau) \propto p_\theta^e(\tau) \exp\left(\beta \sum_{t=1}^{T} g_t(u_t, h_t^e) \right).$$

(The non-additive case follows analogously, replacing g_t with the residual reward attributable to turn t given the history; the per-turn structure of the result is unchanged.)

Conditional distribution of u_t. Fix history h_t^e. The conditional distribution of u_t under $\bar{q}$ is obtained by marginalizing over future continuations:

$$\begin{aligned} \bar{q}(u_t \mid h_t^e) \propto \pi_\theta^e(u_t \mid h_t^e) \exp\big(\beta g_t(u_t, h_t^e)\big) \\ \cdot \, \mathbb{E}_{p_\theta^e} \left[\exp\left(\beta \sum_{s=t+1}^{T} g_s(u_s, h_s^e) \right) \,\middle|\, h_t^e, u_t \right]. \end{aligned}$$

Factoring out the past gives a soft continuation value $V_t(u_t, h_t^e)$. Define

$$V_t(u_t, h_t^e) := g_t(u_t, h_t^e) + \beta^{-1} \log \mathbb{E}_{p_\theta^e}\left[\exp\left(\beta \sum_{s=t+1}^{T} g_s(u_s, h_s^e)\right) \,\middle|\, h_t^e, u_t\right], \tag{9}$$

The conditional distribution becomes

$$\bar{q}(u_t \mid h_t^e) \propto \pi_\theta^e(u_t \mid h_t^e) \exp\big(\beta V_t(u_t, h_t^e)\big). \tag{10}$$

This is the exact per-turn form of the controlled posterior: it has the same exponential weighting as Eq. (4) with the exact continuation value V_t replacing the local utility $\mathcal{U}_\psi^e$.

One-step approximation. Computing V_t exactly requires averaging over all future trajectories under p_θ^e. This is generally intractable for an autoregressive LLM. ARC uses $\mathcal{U}_\psi^e(u_t; h_t^e)$ as a one-step approximation for V_t: the immediate-reward terms in $\mathcal{U}_\psi^e$ approximate g_t, while the information-gain terms approximate the value of diagnosis for later decisions. Substituting $\mathcal{U}_\psi^e$ for G_t in Eq. (10) and restricting normalization to the finite candidate set $\mathcal{C}_t^e$ yields Eq. (4):

$$\pi_{\mathrm{ARC}}^e(u_t \mid h_t^e) = \frac{\pi_\theta^e(u_t \mid h_t^e) \exp\big(\beta \mathcal{U}_\psi^e(u_t; h_t^e)\big)}{\sum_{u' \in \mathcal{C}_t^e} \pi_\theta^e(u' \mid h_t^e) \exp\big(\beta \mathcal{U}_\psi^e(u'; h_t^e)\big)}.$$

B.4 Proof of the local tilt identities

Fix a history $h = h_t^e$ and a finite candidate set $\mathcal{C} = \mathcal{C}_t^e$. Write u for a candidate (m, a) pair, $\pi_0(u) := \pi_\theta^e(u \mid h)$, and $U(u) := \mathcal{U}_\psi^e(u; h)$. Assume $\pi_0(u) > 0$ for all $u \in \mathcal{C}$.

Local objective and its minimizer. Consider the local KL-regularized objective restricted to $\mathcal{C}$,

$$J_t(\pi) := \beta^{-1} \mathrm{KL}(\pi \,\|\, \pi_0) - \mathbb{E}_\pi[U],$$

over distributions π supported on $\mathcal{C}$. Define the local partition function

$$Z_t(\beta) := \sum_{u \in \mathcal{C}} \pi_0(u) \exp(\beta U(u))$$

and the local tilt

$$\pi_\beta(u) := \frac{\pi_0(u) \exp(\beta U(u))}{Z_t(\beta)}.$$

By the same algebraic manipulation as the trajectory-level proof above, specialized to the discrete candidate set,

$$J_t(\pi) = \beta^{-1} \mathrm{KL}(\pi \,\|\, \pi_\beta) - \beta^{-1} \log Z_t(\beta).$$

The unique minimizer is $\pi = \pi_\beta$.

Pairwise odds identity. For any $i, j \in \mathcal{C}$,

$$\frac{\pi_\beta(i)}{\pi_\beta(j)} = \frac{\pi_0(i) \exp(\beta U(i))/Z_t(\beta)}{\pi_0(j) \exp(\beta U(j))/Z_t(\beta)} = \frac{\pi_0(i)}{\pi_0(j)} \exp\big(\beta[U(i) - U(j)]\big).$$

The log-odds ratio is therefore

$$\log \frac{\pi_\beta(i)/\pi_\beta(j)}{\pi_0(i)/\pi_0(j)} = \beta\big[U(i) - U(j)\big].$$

ARC's per-pair reweighting depends only on the utility *difference*, not its absolute level, so adding a constant to $\mathcal{U}_\psi^e$ does not change the policy.

Monotonicity of expected utility in β. Differentiating $\log Z_t(\beta)$ with respect to β,

$$\frac{d}{d\beta}\log Z_t(\beta) = \frac{1}{Z_t(\beta)}\sum_{u\in\mathcal{C}}\pi_0(u)U(u)\exp(\beta U(u)) = \sum_{u\in\mathcal{C}}\pi_\beta(u)U(u) = \mathbb{E}_{\pi_\beta}[U].$$

Differentiating once more,

$$\frac{d}{d\beta}\mathbb{E}_{\pi_\beta}[U] = \frac{d^2}{d\beta^2}\log Z_t(\beta) = \mathrm{Var}_{\pi_\beta}(U) \geq 0,$$

with strict inequality whenever U is nonconstant on the support of π_β. Thus, expected utility is monotonically non-decreasing in β under the local tilt: increasing the inverse temperature concentrates more mass on higher-utility candidates. The deployed controller also uses sparse triggers, including the minimum-utility threshold $\theta_{\min}$, before selecting the candidate with the highest weight under Eq. (4). The identity above concerns the local tilt on a fixed candidate set with fixed utility values. In the notation of Section 5, with $\pi_\beta = \widetilde{\pi}_{\beta,t}$ and $U = U_t$, this identity reads

$$\frac{\partial}{\partial\beta}\mathbb{E}_{\widetilde{\pi}_{\beta,t}}[U_t] = \mathrm{Var}_{\widetilde{\pi}_{\beta,t}}(U_t) \geq 0.$$

B.5 Signed-shift identity

Let Z_t^e be the local normalizer,

$$Z_t^e = \sum_{(m',a')\in\mathcal{C}_t^e}\pi_\theta^e(m',a' \mid h_t^e)\exp\big(\beta\,\mathcal{U}_\psi^e(m',a';h_t^e)\big).$$

Taking logarithms in Eq. (4) gives

$$\log\pi_{\mathrm{ARC}}^e(m,a \mid h_t^e) = \log\pi_\theta^e(m,a \mid h_t^e) + \beta\,\mathcal{U}_\psi^e(m,a;h_t^e) - \log Z_t^e,$$

Subtracting the base-model log-probability gives the signed shift,

$$\Delta_t^e(m,a) := \log\pi_{\mathrm{ARC}}^e(m,a \mid h_t^e) - \log\pi_\theta^e(m,a \mid h_t^e) = \beta\,\mathcal{U}_\psi^e(m,a;h_t^e) - \log Z_t^e. \tag{11}$$

Exponentiating both sides shows that the signed shift is a local density ratio:

$$\exp\big(\Delta_t^e(m,a)\big) = \frac{\pi_{\mathrm{ARC}}^e(m,a \mid h_t^e)}{\pi_\theta^e(m,a \mid h_t^e)}.$$

The summary statistic Δ_t^e is therefore a per-candidate diagnostic of how strongly ARC modifies the base distribution at turn t: positive values indicate up-weighting, negative values indicate suppression, and zero exactly recovers the base policy.

B.6 Information-theoretic interpretation of the utility terms

This subsection gives the information-theoretic meaning of the diagnosis and verification terms in Eq. (3), relates the spread and loop penalties to the trajectory objective, and reports held-out estimates of the full-trajectory decomposition.

Diagnostic information gain. Fix a history h, mode m, and action a. Let F be the active fault variable and O the next observation. By the definition of conditional mutual information,

$$I(F;O \mid h,m,a) = H(F \mid h,m,a) - H(F \mid O,h,m,a).$$

Since F is part of the current repair state *before* choosing (m,a), the controller's prior fault belief at this turn does not depend on the candidate yet to be chosen:

$$H_{q_\phi}(F \mid h,m,a) = H_{q_\phi}(F \mid h).$$

Therefore

$$I(F;O \mid h,m,a) = H_{q_\phi}(F \mid h) - \mathbb{E}_{O\sim p(\cdot\mid h,m,a)}\big[H_{q_\phi}(F \mid h,m,a,O)\big].$$

The diagnostic value term is the expected reduction in fault uncertainty caused by the next observation, conditional on the candidate (m,a). Let S_{t+1} be the next stage state and O_{t+1} the next observation. The same argument gives

$$I(S_{t+1};O_{t+1} \mid h,m,a) = H(S_{t+1} \mid h,m,a) - \mathbb{E}_{O_{t+1}}\big[H(S_{t+1} \mid h,m,a,O_{t+1})\big],$$

The stage-information term is the expected reduction in uncertainty about stage resolution.

Deployed proxy and `blame`. The deployed $\widehat{\Delta H}_F$ assigns every diagnostic command the largest current stage risk, a bounded stand-in for the gain above. `status` and `test` return observations of the active stage, so they can reduce the agent's uncertainty. `blame` returns only a confirmation, so $I(F; O \mid h, m, a) = 0$ for the agent; the recorded fault location instead enters the action channel U. Its value therefore lies in $\mathcal{C}_e^{\mathrm{act}}$: `blame` makes the agent's current fault hypothesis part of the audited record.

Spread and loop penalties. $\Phi_{\mathrm{spr}}(a, h_t^e)$ and $\Phi_{\mathrm{loop}}(a, h_t^e)$ are one-step versions of the trajectory-level penalties $\Phi_{\mathrm{spr}}(\tau)$ and $\Phi_{\mathrm{loop}}(\tau)$ in the ARC objective (counting edited lines outside the believed fault span and wasted turns from stalled test-edit loops, respectively). The spread penalty is computed from the candidate edit scope and whether the action touches high-belief files or unrelated regions. The loop penalty is computed from repeated tests without new edits, repeated failed commands, truncated actions, and empty turns. These penalties lower the scores of candidates that spread edits beyond the inferred fault region or repeat stalled actions.

We evaluated auditors that predict stage, fault, drift, and their joint state on 4,994 states from all 126 tasks, using five folds with disjoint tasks and 1,000 task bootstrap samples. Each auditor receives the final outcome, then observations, and finally actions taken before the evaluated state. The inputs exclude the target labels. The table reports held-out predictive log loss with the outcome alone, the two successive reductions, and the remaining loss. These reductions measure the additional predictive value of observations and actions for the chosen auditor class.

Table B.1: Predictive log losses and successive reductions.

Target	Outcome only	Observation reduction	Action reduction	Remaining loss
Stage	.8482	.8457	.0000	.0025
Fault	1.2434	.8170	.0129	.4135
Drift	.9341	.6975	.1634	.0732
Joint	2.0966	1.4589	.1360	.5017

Each row partitions the loss conditioned on the outcome into the two successive reductions and the remaining loss. The 95% task bootstrap interval for the action reduction is [.1421, .1857] for drift and [.0939, .1742] for the joint state. Observations account for most of the stage and fault loss reduction; including prior actions further reduces the held-out prediction loss for drift.

B.7 Practical realization of the utility terms

Each term in Eq. (3) uses quantities available during inference. The controller tracks unnormalized risk scores for stages, files, and recent action patterns. Each score starts at zero and is updated as $r \leftarrow \lambda_{\mathrm{belief}} r + 1$ for each new observation; a stage that passes is reset to zero. Table B.2 shows how evidence updates these scores and how they guide candidate scoring and trigger decisions.

Table B.2: **Inference-time belief construction.** ARC updates its belief deterministically from observable trajectory evidence and uses it to score candidates and evaluate named triggers.

Belief object	Observable support	Update rule	Use in utility / triggers
Repair mode M_t	{diagnose, revise, verify}; candidate command family; recent action history	Fixed action parser maps shell commands into diagnose/revise/verify	Action type for the diagnostic gain, verification value, and loop indicator
Fault location F_t	Files named in failing test output or error messages	Score $r \leftarrow \lambda_{\text{belief}} r + 1$ for each mention ($\lambda_{\text{belief}} = 0.85$), starting from zero	Spread penalty for edits outside high-risk files; target of diagnostic commands
Stage state S_t	Current stage id, pass/fail counts, status output, and stage-advance events	Score $r \leftarrow \lambda_{\text{belief}} r + 1$ for the failing stage at each turn; a stage that passes is reset to zero	Diagnostic gain and diagnostic trigger (largest stage risk)
Loop/waste state	Empty commands, repeated tests, repeated failed verbs, repeated full commands	Rolling action window plus the same exponential decay as the file and stage state	Loop penalty and hard-reject trigger for actions that repeat stalled behavior

Entropy-drop approximation. The information gain terms use proxies for uncertainty reduction over one step. Diagnostic commands receive the largest current stage risk as value, and other test commands half of it. A test within one turn after an edit receives verification value 1, because it can resolve whether the active stage has advanced. Edits pay the spread penalty $\sum_f \max(0, 1 - r_f)$ over edited files f with file risk r_f. Thus, $\widehat{\Delta H}_F$ and $\widehat{\Delta H}_S$ are deterministic, bounded proxies motivated by the ideal entropy-drop terms. Table D.2 lists the controller weights and thresholds. Each candidate's signed utility breakdown is logged for trigger analysis and term ablations.

B.8 ARC INFERENCE ALGORITHM

Algorithm 1 summarizes the per-turn inference loop used in all ARC experiments. The base model and hyperparameters stay fixed. At each turn, the controller maintains the repair-state belief $q_\phi(z_t \mid h_t^e)$, scores a candidate set, and executes the base choice unless a trigger activates the reweighted policy in Eq. (4). The triggered selection step applies the candidate restrictions described in the main text and renormalizes the candidate weights over that set.

Algorithm 1 Accountable Repair Control (ARC), per-turn inference

Require: Frozen base policy π_θ^e, setting $e \in \{\mathrm{D}, \mathrm{H}\}$, repair-state belief q_ϕ, candidate budget N, candidate and auxiliary sampling temperatures (0.7 and 1), inverse temperature β, acceptance threshold $\theta_{\min}$, utility weights $\Lambda = \{\alpha, \eta, \gamma_{\mathrm{spr}}, \gamma_{\mathrm{loop}}\}$

Update belief: $q_\phi(z_t \mid h_t^e)$ from harness state, recent observations, and last action
Sample candidate pool $\mathcal{C}_t^e = \{u_i = (m_i, a_i)\}_{i=1}^N$ from π_θ^e at temperature 0.7
if token-level logits available **then**
    Read $\pi_\theta^e(m_i, a_i \mid h_t^e)$ from logits
else ▷ API access
    Sample auxiliary pool at temperature 1; set $\pi_\theta^e(u_i \mid h_t^e)$ to the add-one-smoothed frequency of u_i's normalized command in the pool, using proposal frequencies if the pool matches no candidate
end if
for each $u_i = (m_i, a_i) \in \mathcal{C}_t^e$ **do**
    Compute $U_t(u_i) = \mathcal{U}_\psi^e(u_i; h_t^e)$ via Eq. (3) from q_ϕ, candidate touch set, and harness counters
end for
$u_b \leftarrow \arg\max_{u \in \mathcal{C}_t^e} \pi_\theta^e(u \mid h_t^e)$ ▷ ties: earliest proposal
$\mathcal{A}_t \leftarrow \{u \in \mathcal{C}_t^e \setminus \{u_b\} : U_t(u) > U_t(u_b)\}$
if $\mathcal{A}_t \neq \emptyset$ **then**
    $u_a \leftarrow \arg\max_{u \in \mathcal{A}_t} U_t(u)$
    $G_t \leftarrow [\log \pi_\theta^e(u_a \mid h_t^e) + \beta U_t(u_a)] - [\log \pi_\theta^e(u_b \mid h_t^e) + \beta U_t(u_b)]$
else
    $G_t \leftarrow -\infty$
end if
if $G_t > \theta_{\min}$ **or** a named trigger fires (Table B.3) **then**
    Select the allowed candidate with the highest π_{ARC}^e weight as (m_t, a_t) ▷ Eq. (4)
else
    $(m_t, a_t) \leftarrow u_b$
end if
Execute action a_t, observe o_{t+1}^e, log signed shift Δ_t^e and trigger label for telemetry

The algorithm starts each task with a fresh history. It reuses the fixed hyperparameter set $\Lambda \cup \{\beta, \theta_{\min}, N, \theta_{\mathrm{diag}}, k_{\mathrm{diag}}, \rho_{\mathrm{loop}}, \rho_{\mathrm{spr}}\}$ calibrated once on the development split.

Override-trigger formalization. Alongside the minimum-utility trigger $G_t > \theta_{\min}$, ARC fires two further named triggers that mirror specific failure modes; the conditions can overlap, and each turn is logged under the first condition that holds in the listed order. The loop penalty Φ_{loop} is a binary indicator of the stalled patterns in Table B.2, so $\Phi_{\mathrm{loop}} \geq \rho_{\mathrm{loop}} = 1$ rejects any such action, and the spread limit is $\rho_{\mathrm{spr}} = 0.6$.

Table B.3: **Override-trigger formalization.** The three named triggers in Figure 4(a) use inequalities tested in order; the first match wins, and the controller selects the allowed candidate with the highest π_{ARC}^e weight.

Trigger	Activation rule
`hard_reject`	u_b violates the loop or edit-spread budget: $\Phi_{\mathrm{loop}}(u_b, h_t^e) \geq \rho_{\mathrm{loop}}$ or $\Phi_{\mathrm{spr}}(u_b, h_t^e) > \rho_{\mathrm{spr}}$. ARC always overrides.
θ_{diag}	The largest stage risk exceeds θ_{diag} *and* the last k_{diag} turns contain no diagnostic action. ARC selects the best diagnostic candidate or issues a `tracebenchctl` diagnostic command.
$\theta_{\min}$	A higher-utility non-base candidate u_a exists and its utility-adjusted log-odds satisfy $G_t = S_t(u_a) - S_t(u_b) > \theta_{\min}$. ARC removes u_b and selects the remaining candidate with the highest π_{ARC}^e weight.

The reported defaults are $(\theta_{\mathrm{diag}}, k_{\mathrm{diag}}, \rho_{\mathrm{loop}}, \rho_{\mathrm{spr}}) = (0.5, 3, 1.0, 0.6)$, calibrated once on the 16-task development split.

B.9 BELIEF MISSPECIFICATION AND SAFEGUARDS

ARC uses q_ϕ in candidate scores and trigger decisions, so belief errors can affect both. Sparse triggers, hard loop/spread rejects, and cooldowns constrain overrides. Table B.4 lists belief errors, safeguards, and diagnostic checks.

Table B.4: **Implementation safeguards and diagnostic checks for q_ϕ misspecification.**

Possible misspecification	Safeguard in ARC	Observable check
Failing stage unresolved without diagnosis	θ_{diag} trigger promotes diagnosis when stage risk is high and no recent diagnosis exists	Trigger mix and `blame` count in Figure 4(a,b)
Fault belief too concentrated on a stale file	Evidence decay and stage updates reduce stale evidence; edits outside current evidence receive spread penalties and may trigger a hard reject	Spread ablation in Table D.6; diff precision drop when Φ_{spr} is removed
Stage belief stale after a repair	`status`/`test` observations deterministically update stage state; a passed stage is clamped out of the risk state	Stage-progress logs and matched timelines in Appendix E.4
Entropy proxy overrewards diagnosis frequency	Recent-diagnosis window k_{diag}, override cooldown, and the spread and loop penalties discourage repeated diagnosis	scheduled-`blame` and $-\mathcal{U}_{\text{diag}}$ rows in Table D.6
Candidate budget and selection rule	best-of-N and heuristic rerank controls keep ARC's candidate budget fixed and vary the selection rule	Compute-matched controls in Table D.8

C TRACEBENCH-CLI CONSTRUCTION AND TASK DETAILS

This section documents benchmark details that support experimental setup and interpretation.

C.1 SOURCE COLLECTION, REFERENCE SOLUTIONS, AND CLI CONSTRUCTION

TRACEBENCH-CLI starts from programming problems collected directly from the Codeforces website. Codeforces also supplies hard programming tasks for prior executable-code benchmarks (Wang et al., 2025). The retained 126-task manifest has a mean Codeforces rating 2677 and a median rating 2700, with problem metadata, statements, and tags preserved in the released task package.

Table C.1: **Reference-solution acquisition and validation.** Candidate references pass executable validation before retention, with model review providing additional semantic checks.

Source	Artifact produced	Failure handling	Retention check
Qwen3-Coder-480B	Candidate Python reference under retry prompts	Failed candidates are discarded or routed to another solver path	Codeforces-style executable checks plus internal harness assertions
GPT-5.2-Codex	Independent candidate reference or repair of a failed candidate	Same retry-and-discard policy; no candidate is retained by agreement alone	Same executable checks; edge cases are added when failures expose missing coverage
Claude Opus 4.5	Candidate reference and semantic reviewer	Used to inspect hard corner cases and validate human completions	Executable pass remains required after semantic review
Human authors	Completion of the remaining hard cases, roughly 40% of retained references	Human solution is revised until executable validation passes	Claude Opus 4.5 review plus full replay of the generated reference repair script

The validated reference program $P^\star$ is the source of both answers and tests. We parse $P^\star$ to extract subproblems for each function and record their dependency order. We then build staged validation cases from I/O traces at call sites and checks on the complete program. Across the 126-task set this yields 537 named subproblem records, 434 stage gates after the environment stage, and 1,727 hidden assertions. The reference repair script `solution/solve.sh` is generated from $P^\star$: it first repairs the environment fault at stage 0 and then restores the clean reference files. The script is reserved for offline validation and auditing, with access restricted to the evaluator.

Table C.2: **Gold126 construction metadata.** The manifest is the fixed paired-evaluation task list used throughout the paper; counts summarize the released task package before any model evaluation.

Property	Detailed distribution on 126 tasks
Codeforces rating	mean 2677, median 2700, range 2000–3500
Rating bands	2000–2399: 29; 2400–2799: 48; 2800–3199: 31; 3200–3500: 18
Stage gates	total 560; 126 environment gates and 434 post-environment logic gates
Stages per task	3: 13; 4: 56; 5: 46; 6: 10; 7: 1; mean 4.44
Named subproblems per task	total 537; 2: 6; 3: 40; 4: 33; 5: 24; 6: 12; 7: 8; 8: 2; 9: 1
Hidden assertions per task	total 1,727; range 7–29; mean 13.7; most common counts 11: 34, 14: 23, 17: 15
Injected logic faults per task	1: 90; 2: 25; 3: 10; 4: 1; mean 1.38
Environment templates	E1 Makefile path fault: 64; E2 package-initialization typo: 62
Most common tags	math 68, dp 51, combinatorics 39, number theory 27, brute force 23, greedy 22

Table C.3: **Codeforces rating histogram for Gold126.** Ratings match source problem metadata.

Rating	2000	2100	2200	2300	2400	2500	2600	2700	2800	2900	3000	3100	3200	3300	3400	3500
Tasks	7	8	7	7	10	4	17	17	10	6	9	6	6	5	3	4

Faults are injected after $P^\star$ is validated. Logic faults are changes to the abstract syntax tree (AST) that alter program behavior, including boundary shifts, wrong operators or conditions, missing updates, helper misuse, and missing corner cases. For each injected logic fault, we record its function, file, clean span, buggy span, fault family, and a minimal patch restoring $P^\star$. We inject CLI

environment faults separately to create Stage 0 and record the affected files and lines. Multi-fault cases are represented as a list of fault records; offline metrics use the active-stage span or the union of documented spans depending on the analysis.

Table C.4: **Known fault regions.** Recorded during fault injection and validated by replay.

Fault source	Stored provenance	Replay validation	Metric use
E1 environment fault	Makefile line, buggy text, fixed text, reference repair command	Stage 0 must fail before the Makefile repair and pass after replay	Span enters edit-locality audit for build-path repairs
E2 environment fault	Package-initialization line, buggy text, fixed text, reference repair command	Stage 0 must fail before the initializer repair and pass after replay	Span enters edit-locality audit for import-time repairs
AST logic fault	Function, file, clean span, buggy span, fault family, anchor metadata, counterfactual patch	The responsible stage must fail before the patch and pass after replay from $P^\star$	Span supports diff precision, minimal-span ratio, and active-stage attribution
Multiple faults	One provenance record per injected fault, ordered by stage exposure	Individual-failure and setwise-minimality checks reject degenerate cases	Active span is stage-specific; union span is used only for aggregate locality checks

The full generated CLI package contains 128 tasks. Before paired evaluation, we check each task by replaying `solve.sh`, executing hidden stage tests, parsing the recorded fault regions, and checking stage consistency. Two tasks are excluded because replaying their reference repairs is unsuitable for paired attribution evaluation. The resulting Gold126 manifest is used in all paired comparisons and is fixed before paired evaluation begins.

Execution confirmed all 126 buggy workspaces fail and all 126 workspaces repaired by the reference script pass; the 174 recorded fault regions each cover a complete function containing an injected fault. We reconstructed 252 final workspaces and examined all ten passing patches by command deletion and minimization, checking interactions between changes. Four patches require edits outside the original span: deleting those edits makes the suite fail. The secondary analysis expands the accepted region to include those necessary changes and reports adjusted diff precision in Table C.5(a).

All 252 workspace replays completed. Regression counts and differential agreement are reported in Table C.5(b). Differential testing used 200 fresh inputs per passing task and 1,000 per condition.

Table C.5: **Alternative repairs and execution checks.** (a) Ten passing patches; four require edits outside the original span. The complete suite passes after removing deletable commands. (b) Regression counts from 252 workspace replays and agreement on 1,000 fresh inputs per condition.

(a) Passing patches

Measure	Base	ARC
Adjusted diff precision	.900	.900
After minimization	1.000	1.000
Deletable commands	21	9

(b) Execution checks

Measure	Base	ARC
Regressions	3	3
Reference agreement	993/1000	1000/1000

C.2 TASK-LEVEL STATISTICS

TRACEBENCH-CLI contains 126 hard programming-completion tasks derived from Codeforces-style problems, each packaged as a Docker image with a persistent workspace, the full problem statement, a faulty implementation, a staged test harness, and the `tracebenchctl` diagnostic tool. All paired comparisons in the main text use the 126-task manifest under a single shared budget.

C.3 STAGE AND INJECTED-BUG DISTRIBUTION

Every task has stages that must be passed in order. Stage 0 checks environmental consistency (module paths, package imports, and generated source placement); later stages validate logical correctness through hidden tests. Across the 126-task set, the stage count per task ranges from 3 to 7, and the

number of injected logic bugs ranges from 1 to 4. Most tasks have 4 to 5 stages with 1 logic bug. Stage 0 is always present, so environment failures are recorded separately from logic failures.

Why the distribution matters for ARC. Two structural properties of the distribution shape ARC's behavior. First, the variation in stage count means the controller cannot rely on a fixed budget per stage; instead, the diagnostic gain and trigger use the risk of the current stage. Second, the variation in injected-bug count means the action-side control problem has a non-trivial range of difficulty within the 126-task manifest, which is why the strong-model rows in Figure 5 show different per-base improvement magnitudes.

C.4 Environment fault templates

The benchmark uses documented injected faults to evaluate fault attribution throughout each repair. Template *E1* modifies the project's Makefile so that a generated Python source file is written to the wrong path under `src/`, breaking downstream import. The agent must locate the path mismatch in the build pipeline. Template *E2* introduces a fatal typo in a package-initialization file; the agent must find a small import fault that can be easy to miss when reading the file. Later stages contain logic faults injected directly into the task implementation. The recorded fault regions cover these three sources and support the diff precision and minimal-span metrics, and each is documented in the per-task manifest released with the benchmark.

C.5 The fixed 126-task manifest

Candidate tasks that the authors could not validate against all stage assertions were excluded before any paired evaluation. The released 126-task manifest is the canonical task set used throughout the paper, with task identity, budget, and evaluation harness held fixed across paired comparisons.

C.6 `tracebenchctl` interface

The tool in each container exposes four commands: `init` (harness only), `status`, `test`, and `blame`. Only the last three are available to the agent during evaluation. `status` reports progress in the current stage; `test` executes the stage tests without revealing hidden test contents; `blame` records a fault location named by the caller and returns a confirmation; each call is logged as an explicit diagnosis. Offline diff precision and minimal-span ratio use the known fault regions. The agent and ARC use visible task evidence.

`blame` records. A call `tracebenchctl blame FILE:START-END --note NOTE` appends the location, the note, and a timestamp to the task's blame log and prints `Recorded blame: FILE:START-END`. The command returns no information about the fault regions or the hidden tests; it makes the caller's current fault hypothesis explicit and countable. When an override needs a command of its own, ARC records a `blame` on the file with the highest risk score, or, if that command was issued recently, reads the file or calls `status`.

Logging and reproducibility. Every invocation of `tracebenchctl` (including `blame`) is logged with timestamp, command arguments, agent input window snapshot, and command output. The logs support trajectory replay and recomputation of all metrics for individual turns.

C.7 Released trajectory examples for benchmark construction

The release includes raw trajectory logs in addition to the paired ARC runs. Table C.6 relates three such trajectories to the benchmark structure: an environment gate, named subproblem gates, hidden assertions, and known fault regions in specific files.

Table C.6: **Released trajectory examples.** These examples illustrate why the stage-gated CLI format yields interpretable process data: each trajectory can be mapped to a concrete source problem, construction record, active stage, and repair span.

Task	Source problem	Construction profile	What the trajectory exposes
`tb_hard_00000`	Codeforces 1054H, rating 3500; FFT / CRT / number theory	E1 environment fault plus one logic fault; 3 stages, 8 hidden assertions	The agent repairs the build-path fault, passes the helper-level gate, and then fails the full-I/O stage. A pass/-fail label alone would mark failure; the trace shows separate environment repair, partial algorithmic progress, and the remaining active span.
`tb_hard_00012`	Codeforces 571E, rating 3200; geometric progressions	E1 environment fault plus one loop-bound logic fault; 6 stages, 11 hidden assertions	The trajectory passes factorization, power, and progression-intersection gates before failing the active subproblem. A one-line loop-bound edit then clears the remaining stages, showing how staged tests localize a late semantic fault.
`tb_hard_00004`	Codeforces 1190F, rating 3400; probabilistic number theory	E1 environment fault plus three logic faults across modular arithmetic and factorization helpers; 7 stages, 23 hidden assertions	The log separates import repair from the first modular-arithmetic failure and records a broad speculative rewrite before termination. The active-stage span identifies which of the candidate helpers contains the current fault.

D EXPERIMENTAL PROTOCOL, CONTROLS, AND SUPPLEMENTARY RESULTS

This section specifies the benchmarks, models, execution budgets, controller settings, and statistical methods used in the experiments.

D.1 BENCHMARKS USED IN THIS PAPER

The paper uses one primary benchmark (TRACEBENCH-CLI, our own contribution) and three external benchmarks for the cross-benchmark analysis in Figure 2.

TRACEBENCH-CLI (primary, ours). 126 Codeforces-derived programming-completion tasks, each packaged as a Docker container with a test harness and a three-command agent-visible diagnostic tool (`status`/`test`/`blame`) plus a harness-only `init`. Every task is stage-gated (3–7 stages) with 1–4 injected logic bugs and documented environment-fault templates (E1 Makefile mis-routing; E2 package-initialization typo). The 126-task manifest is used for all paired comparisons. Full construction details are in Appendices C.1–C.6.

AgencyBench (Li et al., 2026c). A benchmark in which a model uses a fixed set of tools to generate code and edit patches. We use its specified turn budget in Figure 2. We report its pass rate alongside our traceability proxy computed from the action logs.

TerminalBench (Merrill et al., 2026). ~ 100 realistic command-line tasks requiring the agent to operate in a shell, inspect files, and run real programs. Used in Figure 2 and in a supplementary long-horizon check reported below. Uses the benchmark's native budget; pass rate and trajectory-log proxies are computed analogously.

LongCLI Bench. A longer-horizon CLI benchmark emphasizing persistent workspaces and multi-file edits. Used in Figure 2 to stress-test the outcome-versus-proxy relation under native budgets.

Related evaluation settings. HumanEval, MBPP, APPS, CodeContests, SWE-bench (Jimenez et al., 2024), InterCode (Yang et al., 2023), and SWE-agent (Yang et al., 2024) cover code generation, interactive execution, and repository repair.

External deployment of ARC. On external benchmarks, ARC uses the available tools to diagnose, revise, and verify. Diagnostic actions include file inspection, `grep` and traceback reading, and the benchmark's native diagnostic tools when present; revision actions include edit/write tool calls; verification actions include test or run commands. The fault belief $q_\phi(F_t \mid h_t)$ is maintained over touched files and directories using failed-command evidence and recent inspections. The θ_{diag} trigger counts recent inspections as diagnostic actions. Spread and loop budgets carry over unchanged because they are computed directly from the trajectory. Focus, scope, and stability summarize completed trajectories for evaluation. Transfer results use paired runs on identical task lists and the budgets specified by each benchmark, with the controller's hyperparameters fixed at the values chosen on the development split.

D.2 MODELS USED IN THIS PAPER

All base models are frozen; ARC never updates model parameters. Models are accessed either via commercial APIs (OpenAI, Anthropic, Together) or via local inference (Hugging Face Transformers, vLLM). Table D.1 records the role of each model.

Main paired roles. The two primary paired comparisons in Figure 5 use **Qwen3-32B** (direct paired; local inference chosen so that logits for the ARC utility are available) and **gpt-4o-mini** (harness paired; API-served through the CLI harness). The ablation in Table D.6 also uses gpt-4o-mini. The strong-model paired robustness block in Figure 5 extends the harness-paired protocol to GPT-5.4, GPT-5.4-mini, and GPT-5.4-nano, with fixed controller hyperparameters.

Table D.1: **Models used in the paper.** Tiers follow the cross-benchmark axis of Figure 2. "Role" records the position of each model in the experimental body.

Model	Tier	Access	Role in this paper
GPT-5.4-nano	small	OpenAI API	cross-benchmark (Fig. 2); strong-model paired robustness (Figure 5, 5.4-nano)
gpt-4o-mini	medium	OpenAI API	**main harness paired** (Figure 5, 4o-mini); ablation reference (Table D.6); cross-benchmark (Fig. 2); diagnostic trajectories (Fig. 3)
Qwen3-32B	medium	local HF, bf16, 4×H200	**main direct paired** (Figure 5, Q-D and Q-H); cross-benchmark (Fig. 2)
GLM-5	large	Together API	cross-benchmark (Fig. 2); diagnostic trajectories (Fig. 3)
GPT-5.4	large	OpenAI API	cross-benchmark (Fig. 2); strong-model paired robustness (Figure 5); diagnostic trajectories (Fig. 3)
MiniMax-M2.5	large	MiniMax API	cross-benchmark (Fig. 2)
GPT-5.4-mini	reasoning	OpenAI API	cross-benchmark (Fig. 2); strong-model paired robustness (Figure 5)
Claude Opus 4.6	reasoning	Anthropic API	cross-benchmark (Fig. 2); cross-LLM ARC transfer (Figure 5c; Table D.13)
Llama-3.3-70B-Instruct	large	Together API	cross-LLM ARC transfer (Figure 5c; Table D.13)
Qwen2.5-Coder-14B	small	AWQ endpoint	matched-sampling selection rules (Table 3a; Table D.9)
o4-mini	reasoning	OpenAI API	diagnostic trajectories (Fig. 3); action shares (Figure D.6)
Qwen-3.5 (Qwen3.5-397B-A17B)	—	Together API	action shares (Figure D.6)

Cross-benchmark role. Figure 2 uses eight models (GPT-5.4-nano, gpt-4o-mini, Qwen3-32B, GLM-5, GPT-5.4, MiniMax-M2.5, GPT-5.4-mini, Claude Opus 4.6) in an outcome-versus-traceability-proxy scatter across AgencyBench, TerminalBench, and LongCLI Bench under each benchmark's native budget. These runs establish the baseline outcome–process mismatch. Figure 5(c) additionally evaluates ARC controller transfer on gpt-4o-mini, Llama-3.3-70B-Instruct, and Claude Opus 4.6; Llama-3.3-70B-Instruct is included only for that controller-transfer robustness check.

Comparison design. Figure 2 compares outcomes and process scores under native budgets. Figure 5 compares each model with and without ARC, fixing tasks and controller settings.

D.3 Execution settings and backends

The experiments use three realized backends. First, the direct comparison (DIRECT vs. DIRECT+ARC) is run locally with Qwen3-32B using Hugging Face Transformers in bfloat16 on 4×H200 GPUs. Second, the main controller experiments on the 126-task manifest use a frozen API-served model (gpt-4o-mini) through a Docker-backed harness that executes every agent command inside a persistent task container. Third, we also run cross-benchmark external runs and strong-model checks with frozen API backends, primarily through OpenAI and Together. The matched-sampling selection-rule comparison (Table 3(a), Table D.9) uses a frozen Qwen2.5-Coder-14B AWQ endpoint. All main-text and appendix results use frozen base models; ARC acts only as an inference-time controller.

D.4 Budgets and decoding

The main TRACEBENCH-CLI protocol uses a short budget of 20 turns, 80 tool calls, 32K input tokens, and 4K output tokens per turn. Auxiliary long-budget checks use 50 turns, 200 tool calls, and 64K input tokens. Direct prompt-only baselines use temperature 0.2. ARC samples its candidate pool $\mathcal{C}_t^e$ at temperature 0.7 with $N = 3$ proposals per turn; under API access, N further proposals are sampled at temperature 1. Candidates are the distinct proposals after whitespace normalization, and

$\pi^e_\theta(m, a \mid h^e_t)$ in Eq. (4) is the add-one-smoothed frequency of each candidate's normalized command among the auxiliary samples; if no auxiliary sample matches a candidate, proposal frequencies are used instead. External benchmarks (AgencyBench, TerminalBench, LongCLI Bench) follow each benchmark's native turn and tool-call caps; the controller uses the in-domain protocol's fixed hyperparameters.

D.5 CONTROLLER HYPERPARAMETERS

The controller settings were selected once on 16 development tasks from the same Codeforces pool as the evaluation tasks. This split was reserved for parameter selection and was disjoint from the 126-task evaluation manifest. We then kept these settings fixed across all models, budgets, and benchmarks. Table D.2 consolidates every symbol that appears in the controller derivation and Algorithm 1, including the non-base trigger $G_t = [\log \pi_\theta(u_a) + \beta U_t(u_a)] - [\log \pi_\theta(u_b) + \beta U_t(u_b)]$, with $U_t(u_a) > U_t(u_b)$.

We assessed sensitivity around these fixed defaults in seven runs, each varying one hyperparameter on 42 tasks. All runs used six calls per turn and completed without errors. Moderate changes preserved similar stage progress, while aggressive gate or reweighting settings reduced it. Scaling each weight from 0.5 to 1.5 times its default on the logged candidates preserved the top choice on 99.2% of turns and pairwise orderings on 98.7%.

Table D.2: **Full ARC hyperparameter table.** All values are calibrated once on the 16-task development split and reused without retuning across every paired and external comparison.

Symbol	Value	Role
Belief / candidate sampling		
λ_{belief}	0.85	decay on the repair-state belief update $q_\phi(z_t \mid h_t)$
N	3	candidate count per turn (size of $\mathcal{C}_t$)
candidate temperature	0.7	candidate sampling temperature
auxiliary temperature	1.0	auxiliary base-probability estimate temperature (API access)
Trigger / activation		
β	0.5	inverse temperature in Eq. (8)
$\theta_{\min}$	-0.5	log-odds trigger threshold (negative; allows a higher-utility non-base choice with slight base-prob loss)
Utility weights ($\mathcal{U}_\psi$ in Eq. 3)		
α	1.0	fault-information-gain coefficient
η	0.5	stage-information-gain coefficient
γ_{spr}	0.7	spread penalty Φ_{spr}
γ_{loop}	1.0	loop penalty Φ_{loop}
Override triggers		
θ_{diag}	0.5	diagnostic-trigger threshold on the largest stage risk
k_{diag}	3	recent-diagnosis window (no diagnostic action in the last k_{diag} turns)
ρ_{loop}	1.0	hard loop-budget threshold
ρ_{spr}	0.6	hard spread-budget threshold

The diagnostic override fires when the largest stage risk exceeds θ_{diag} and the last k_{diag} turns contain no diagnostic action; the negative $\theta_{\min}$ allows ARC to spend a small amount of base log-odds for a higher-utility non-base action, while the requirement $U_t(u_a) > U_t(u_b)$ prevents saturation by trivially positive gaps. The strong-model rows in Figure 5 (GPT-5.4, GPT-5.4-mini, GPT-5.4-nano) and the cross-LLM rows of Figure 5(c) reuse Table D.2 unchanged, evaluating transfer under fixed controller settings.

D.6 STATISTICAL METHODOLOGY

We compare solve rates with McNemar's paired test and process metrics with percentile bootstrap ($B = 1000$ resamples, 95% CIs). Bold entries in Figure 5 and Table D.6 mark the better paired value: higher Solve / DIFF / M-Span / Blame and lower TWE / Repeat / Waste.

Table D.3: **Compact sensitivity map for the main ARC controls.** The table records the role of each scalar control and the paper location that checks the corresponding failure mode.

Control	Default	Primary sensitivity	Evidence / check
β	0.5	Strength of the exponential tilt; too small approaches the base policy, too large makes utility dominate base preference	Utility-gap telemetry in Figure D.3; term ablations in Table D.6
$\theta_{\min}$	-0.5	Override aggressiveness; negative threshold permits limited base-probability loss only when $U_t(u_a) > U_t(u_b)$	Trigger partition in Figure 4(a); non-base improvement condition prevents saturation
N	3	Candidate diversity and extra proposal budget	best-of-N and heuristic controls in Tables D.7–D.8
Candidate temperature	0.7	Diversity of candidate proposals before scoring	Same candidate temperature is used by ARC and the compute-matched selection rules
Auxiliary temperature	1.0	API-only empirical estimate of base preference over the candidate set	Same auxiliary pool is used by best-of-N, heuristic rerank, and ARC in Table D.7
λ_{belief}	0.85	Memory of stage/file/action evidence in q_ϕ	Belief construction and stale-evidence safeguards in Appendices B.7–B.9

For the Qwen3-32B direct pair in Figure 5, exact repeats average 0.33 and 0.36 per task under DIRECT and DIRECT+ARC. These are roughly 10% of mean Waste (3.30 and 3.60). Test-without-edit dominates the remaining Waste; the run log records truncated and empty turns separately.

Bootstrap CIs for the strong-model block. Table D.4 gives bootstrap 95% CIs (B=1000 paired resamples) for Figure 5(a). All three GPT-5.4 models have Waste, Repeat, and `blame` deltas with CIs excluding 0.

Table D.4: **Strong-model paired deltas with 95% bootstrap CIs ($B = 1000$) on the 126-task manifest.** For Waste and Repeat, CIs below 0 indicate improvement; CIs above 0 indicate higher diff precision or more diagnostic calls. Numbers match Figure 5(a).

Base	ΔDIFF	ΔWaste ↓	ΔRepeat ↓	Δ`blame`
GPT-5.4	$+0.05$ $[+0.02, +0.08]$	-1.67 $[-2.10, -1.23]$	-1.12 $[-1.39, -0.86]$	$+2.91$ $[+2.48, +3.32]$
GPT-5.4-mini	$+0.08$ $[+0.05, +0.11]$	-3.47 $[-3.95, -2.94]$	-1.81 $[-2.10, -1.51]$	$+2.73$ $[+2.30, +3.13]$
GPT-5.4-nano	-0.18 $[-0.24, -0.11]$	-8.37 $[-9.10, -7.55]$	-4.22 $[-4.61, -3.83]$	$+3.42$ $[+2.96, +3.85]$

D.7 REPRODUCIBILITY POINTERS

All main-text figures and tables can be reconstructed by the asset-generation scripts under the experiment root. The paired 126 API run contains 126 harness JSON trajectories and 126 ARC trajectories under the main run directory; supplemental ablation and transfer trajectories live under the supplement directory; the GPT-5.4-family check is stored under the strong-model directory; and matched case-study traces are preserved turn-by-turn under the case-study subtree.

D.8 CROSS-BENCHMARK PROXY VIEWS

Figure 2 shows that outcome scores are weakly correlated with the process proxies on AgencyBench, TerminalBench, and LongCLI Bench. Figure D.1 shows the composite proxy in its three components.

Figure D.1 shows the individual components of the composite proxy in Figure 2. Each row plots one proxy component against outcome score across the three external benchmarks, and all nine panels show the same qualitative pattern: shallow fitted slope, large vertical spread at every outcome level, and substantial between-benchmark dispersion for the same model. Focus has the strongest correlation with outcome and shows substantial variation at each outcome level. Scope and stability correlate weakly with outcome on AgencyBench and LongCLI Bench. These panels show that similar outcome scores can accompany different process profiles.

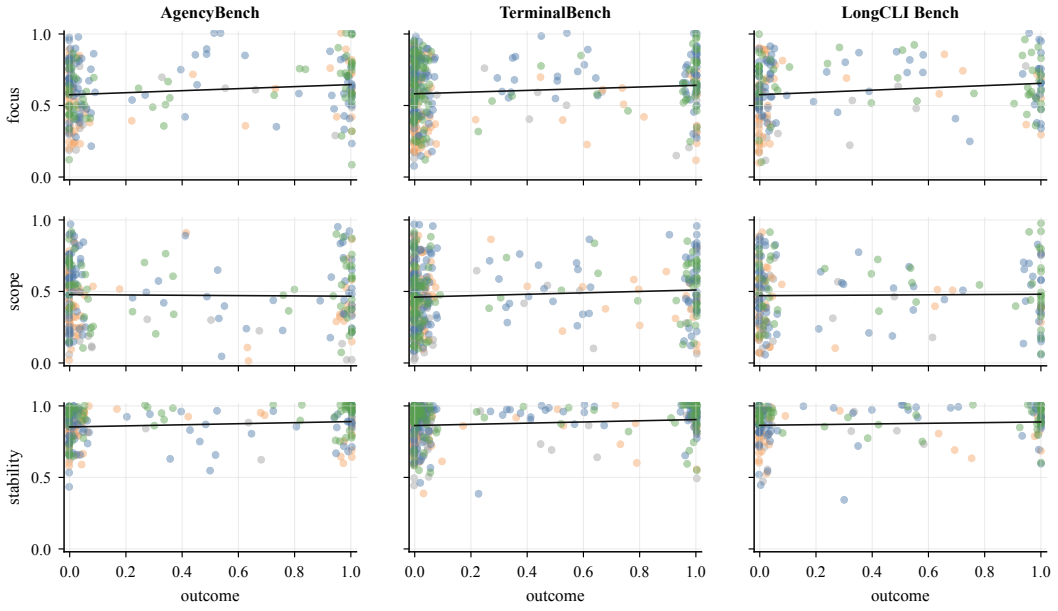


Figure D.1: **Component-wise proxy breakdown.** Each row plots one proxy component (focus, scope, stability) against outcome across the three external benchmarks.

D.9 ADDITIONAL DIRECT DIAGNOSTICS FOR TRACEBENCH-CLI

To explain the results in Figure 3 and Figure 4, we compare runs that stop at Stage 0 with those that advance further (Figure D.2), examine when ARC changes the base action (Figure D.3), and measure changes for each paired task (Figure D.4).

Stage-0 pressure versus later repair. Figure D.2 partitions the 200 trajectories of Figure 3 by whether they ever advance beyond Stage 0 (teal) or remain there (red). Across all four process axes (diff precision, blame, wasted turns, and stage progress), the teal subset still exhibits substantial interquartile spread, and on blame and wasted turns its medians differ from the red subset by less than one standard deviation. Process variation therefore remains after environmental repair.

Controller decisions. Figure D.3 reports when the controller overrides a proposed action. Panel (a) groups turns by diagnosis plus verification utility and diffusion plus waste penalty. Color shows the fraction of turns in each bin where the controller overrode the proposal. Panel (b) reports trigger frequencies: hard rejections are most frequent, followed by diagnostic threshold and minimum utility triggers. Panel (c) compares mean utility terms for accepted and overridden turns. The panels relate override decisions to the utility components and named triggers.

Blame locations. Table D.5 checks every `blame` location in the gpt-4o-mini HARNESS pair against the recorded fault regions used for diff precision, and separates the attributions issued by the model from those that ARC records when it overrides.

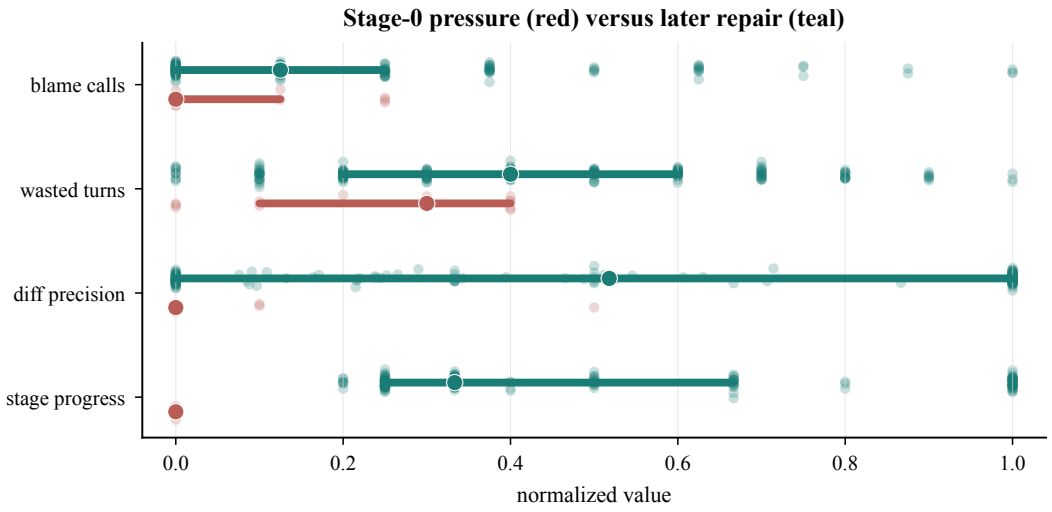


Figure D.2: **Stage-0 pressure versus later repair.** Red points are trajectories trapped in Stage 0; teal points advance further. Thick lines show interquartile ranges with median dots. Substantial variation in blame and waste persists beyond the environment stage.

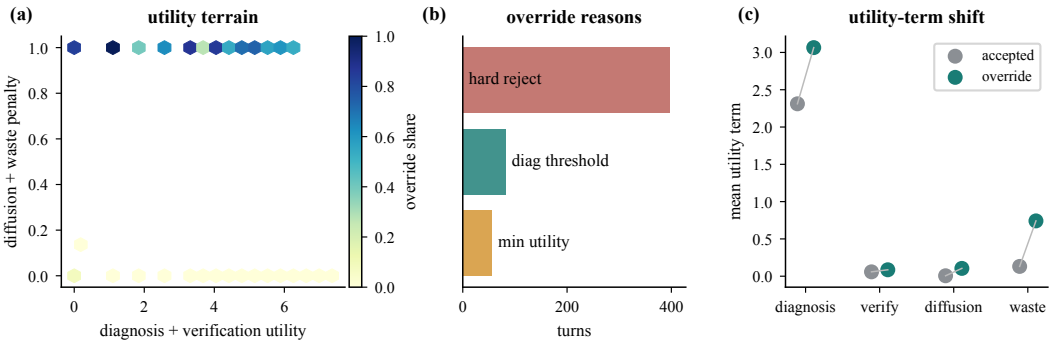


Figure D.3: **ARC controller logs.** (a) Override frequency across diagnosis plus verification utility and diffusion plus waste penalty. (b) Trigger counts for hard rejections, diagnostic thresholds, and minimum utility. (c) Mean utility terms for accepted and overridden turns.

Table D.5: **`blame` locations against recorded fault regions (gpt-4o-mini HARNESS pair, 126 tasks).** File hit: the named file contains a recorded fault region. Line overlap: the named line range intersects the region. Lines in region: share of named lines inside the region, averaged over calls. Range: median number of named lines.

Source	Calls/task (total)	File hit	Line overlap	Lines in region	Range
HARNESS, issued by the model	2.48 (312)	79.2%	16.7%	2.6%	1
HARNESS+ARC, issued by the model	2.08 (262)	59.5%	27.9%	5.7%	21
HARNESS+ARC, recorded by ARC	3.12 (393)	63.1%	63.1%	1.4%	160

Paired changes. Figure D.4 complements the aggregate comparison on 126 tasks in Figure 4 with distributions of paired changes. Panel (a) shows standardized paired changes for each metric (each violin centered at zero on a $[-1, 1]$ axis). Panel (b) reports bootstrap effect sizes as horizontal error bars. Panel (c) relates the change in diagnostic calls to waste reduction for each paired task.

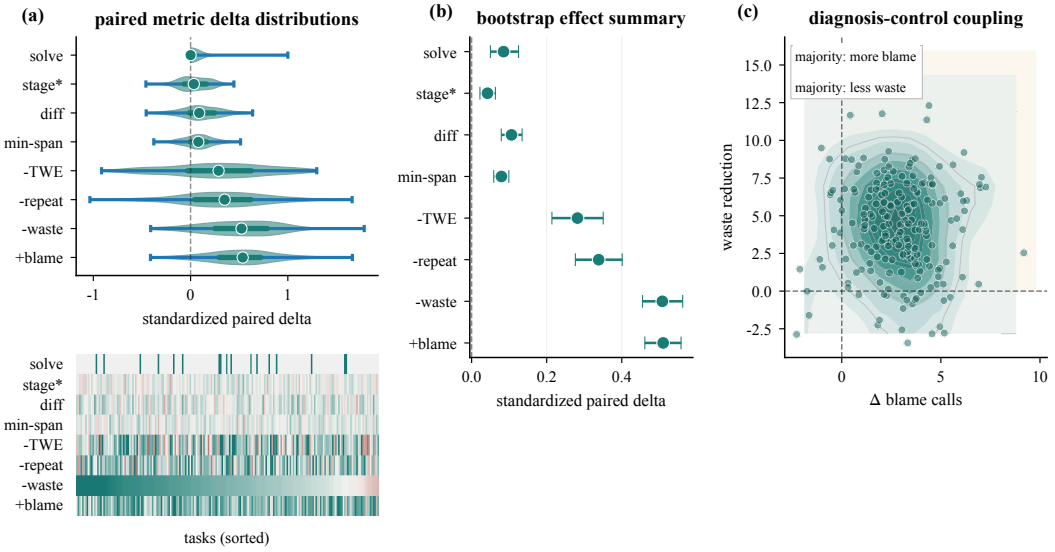


Figure D.4: **Changes for the 126 paired tasks (aggregate means in Figure 4).** (a) Standardized changes in each metric. (b) Bootstrap effect sizes. (c) Change in diagnostic calls against reduction in wasted turns; the annotated region contains most task pairs.

D.10 Distributional evidence for the joint utility

Failures occur at different combinations of process quality and task outcome, and the logs show where ARC intervenes. Figure D.5 reports four analyses of the trajectory logs: process metrics for each trajectory (panel a), overlap of process failure types across tasks (panel b), Spearman correlations among observable process metrics for the repair state in Eq. (2) (panel c), and controller decisions at each turn, corresponding to the override frequencies in Figure 4(a) (panel d).

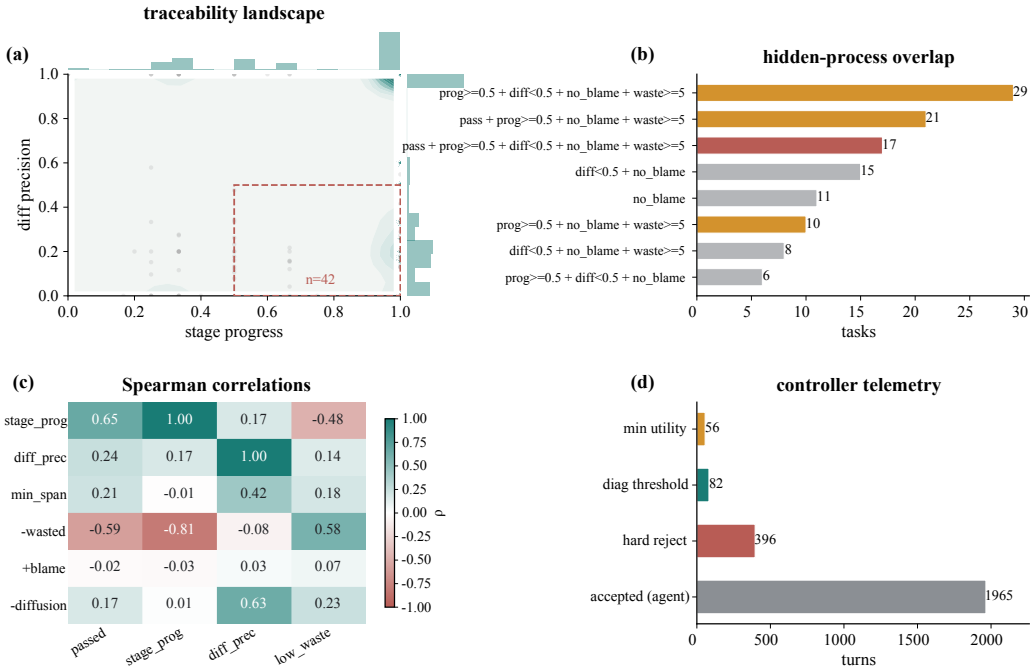


Figure D.5: **Process metrics and controller decisions.** Panels (a)–(c) use Qwen3-32B HARNESS trajectories on the 126 tasks; panel (d) uses the gpt-4o-mini HARNESS+ARC run. Panel (a): stage progress against diff precision; the dashed box marks trajectories with high progress and low precision. Panel (b): per-task overlap of hidden-process failure modes (`diff<0.5`, `waste>=5`, `prog>=0.5`, ...). Panel (c): Spearman correlations between outcome and process metrics. Panel (d): per-turn controller telemetry across its 2,499 logged turns, partitioned into `accepted (agent)`, `hard reject`, `diag threshold`, and `min utility` (override triggers in Figure 4a).

D.11 ACTION UNCERTAINTY ACROSS COMPLETED MODELS

Figure D.6 summarizes logged action shares for each completed model. Commands are grouped into inspect, edit, test, blame, status, exec, and other actions. Each trajectory is divided into early, middle, and late thirds by turn index. Each row reports one model and its completed-run count.

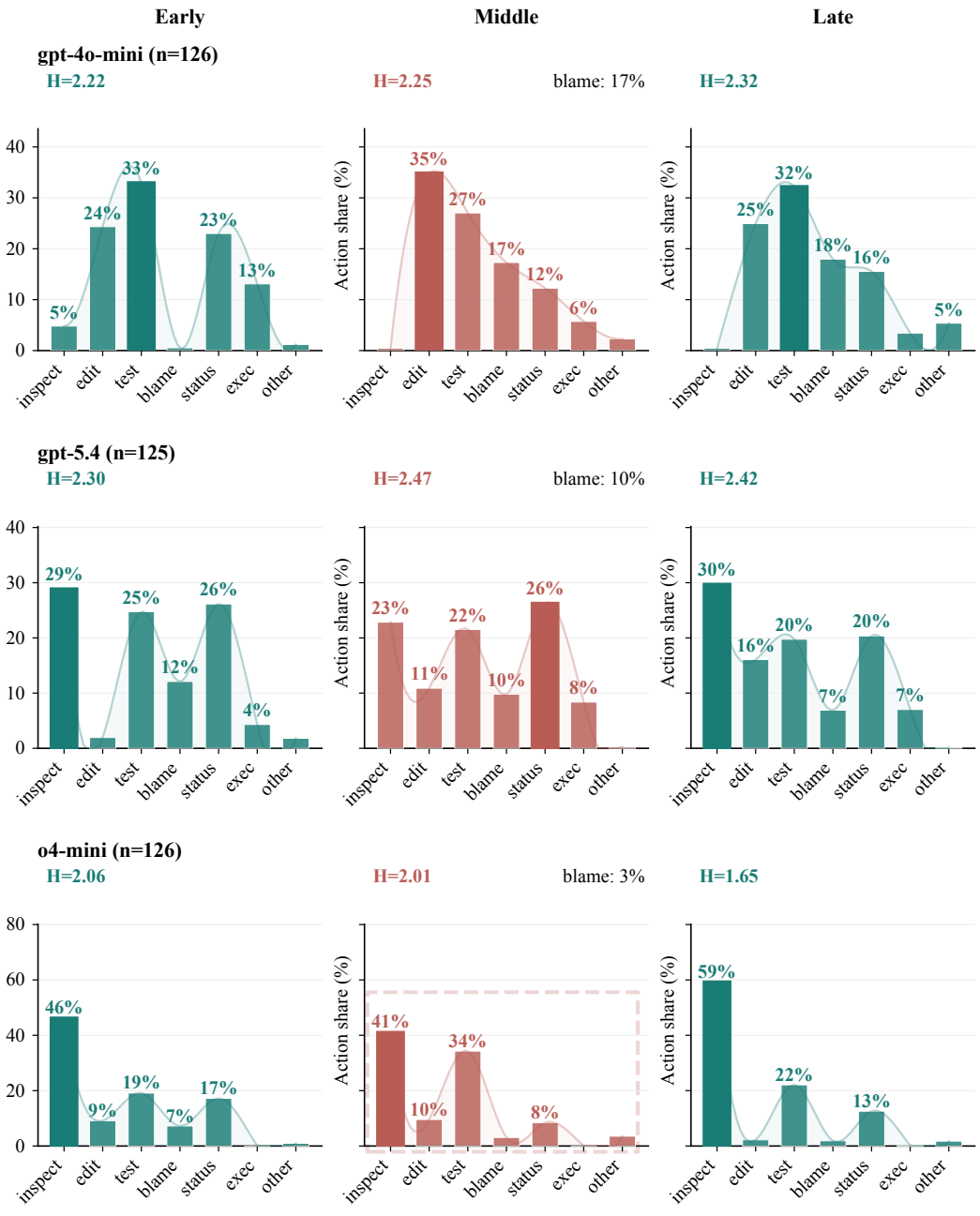


Figure D.6: **Action shares along repair trajectories.** Columns show the early, middle, and late thirds of each trajectory. Bars give the frequencies of seven action categories; H is the entropy of these frequencies. The remaining models appear on the following page.

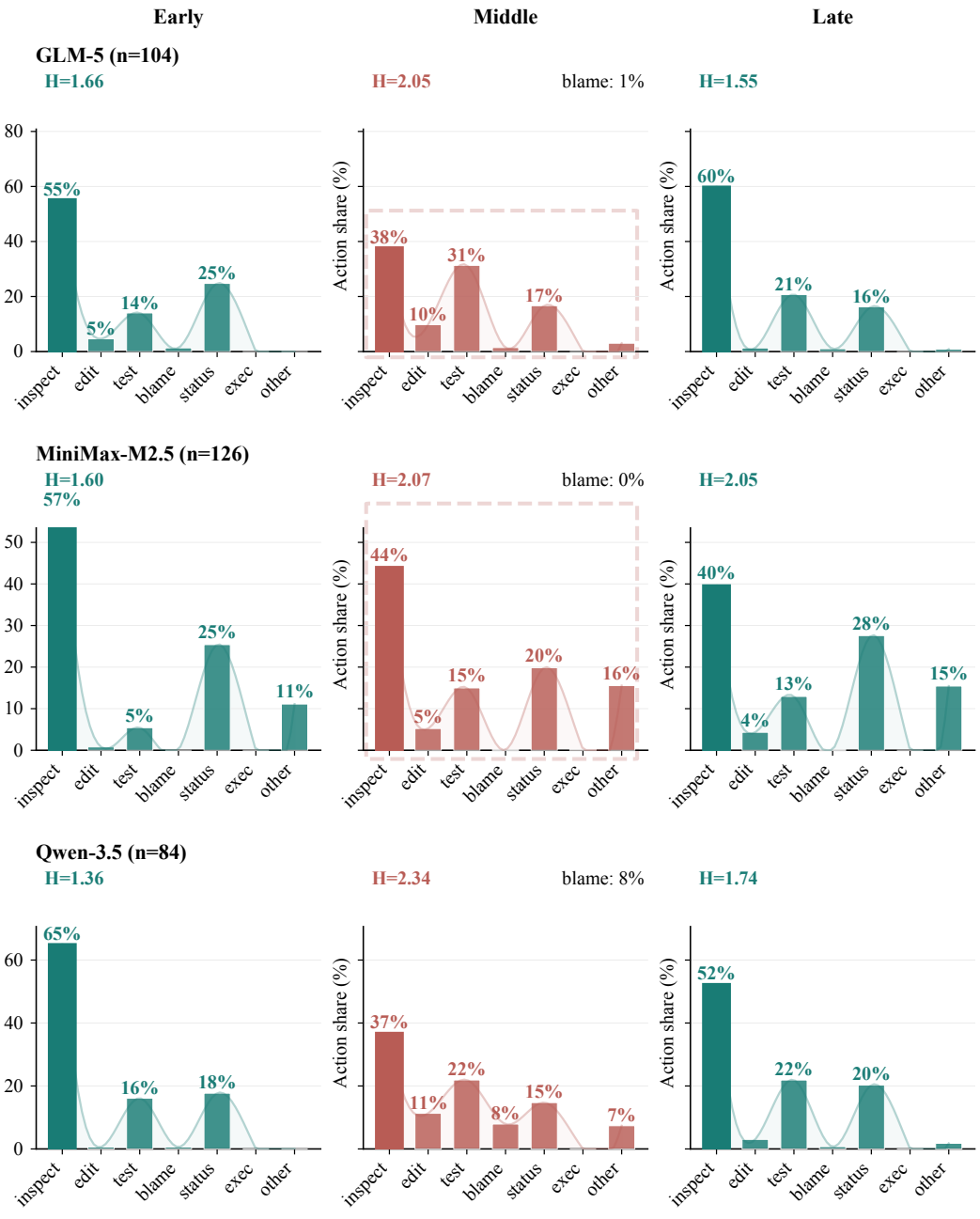


Figure D.6 (continued): Action shares for the other three models; same intervals and categories.

D.12 FULL ABLATION TABLE

Table D.6 expands the Figure 4(d) ablation into four comparisons on the 126-task gpt-4o-mini harness pair. Each ablation row removes one utility group while holding the other utility terms fixed at their main-controller values. The ablation compares the behavioral effects of the utility groups through process metrics (DIFF, Waste, Repeats, Blame).

Table D.6: **Ablations of the diagnosis, spread, and loop terms on the 126-task manifest (gpt-4o-mini, harness).** Each ablation row removes one term, keeping the others at their default values. The first two rows show the unconditioned base and full controller. Bold marks improvements over HARNESS.

Variant	DIFF↑	Waste↓	Repeats↓	Blame↑
HARNESS (base)	0.648	6.18	2.10	2.48
HARNESS+ARC (full)	**0.798**	**1.34**	**0.00**	**5.20**
$-\mathcal{U}_{\text{diag}}$	0.78	1.55	0.00	2.80
$-\Phi_{\text{spr}}$	0.62	1.60	0.00	5.00
$-\Phi_{\text{loop}}$	0.79	4.80	1.10	4.90
scheduled-`blame`	0.69	4.50	1.20	3.80

Three patterns are stable across the table. Removing $\mathcal{U}_{\text{diag}}$ keeps locality and waste close to the full controller but drops `blame` by roughly half. The clearest effect of this term is increased diagnosis frequency. Removing Φ_{loop} preserves diagnosis but increases the number of wasted turns from 1.34 to 4.80, so locality and diagnosis alone do not prevent stalled retesting. Removing Φ_{spr} keeps waste low but drops diff precision from 0.798 to 0.62 while `blame` stays high, so locality and waste require separate control. The scheduled-`blame` variant fires a diagnosis at fixed intervals and recovers part of the full controller's gains. This comparison measures the combined effects of candidate selection, proposal budget, and diagnosis frequency.

D.13 COMPUTE-MATCHED CONTROL SUITE

Table D.7 defines the control suite used to separate proposal budget from utility-based selection. The main comparison is execution-budget matched: all rows use the same task list, harness interface, turn cap, tool-call cap, and token cap. The compute-matched rows also use the same $N = 3$ candidate pool as ARC; under API-only access, they also use the same auxiliary N-sample pool to estimate base preference. These rows differ only in the selection rule for the finite candidate set.

The key comparison is between best-of-N base-prob and HARNESS+ARC: both spend the same proposal budget, but only HARNESS+ARC uses the ARC utility. The heuristic rerank row tests whether a small hand-coded rule set is enough to recover the effect. The scheduled-`blame` row measures how much of the gain fixed diagnostic intervals recover, while the term ablations keep proposal cost fixed and remove one utility channel at a time.

Table D.8 reports the OpenAI comparison in manifest order, with up to 10 attempted tasks per row. GPT-5.4-nano with ARC completed 9 of 10 attempts; GPT-5.4 with heuristic rerank completed 7 of 9; all other rows completed 10 of 10. Timeout or policy-filter failures were retained as errors. Full 126-task solve rates appear separately in Table D.11. The pilot summary uses completed trajectory records; `compute_runs/compute_matched_cost_audit.csv` retains aggregate API usage logs.

We executed 965 distinct proposal branches from 400 states verified by paired workspace replay. All selectors received identical workspaces, histories, proposals, auxiliary samples, model probabilities, and interfaces, and execution provided the outcome labels. Table D.10 counts, for each outcome type, the states in which a selector chose such a candidate among the states where one was available. Only 6 of the 400 pools contain a one-step solving candidate. On this fixed pool, ARC selects more correct diagnoses, improving edits, and useful verifications than the base, with less waste, and captures the same 5 of 6 solving candidates. The table also reports stage-advancing commands and regret for every selector.

Table D.7: **Compute-matched controls for isolating ARC's selection rule.** All rows share the same task manifest and execution budget. Rows marked compute-matched use the same candidate and auxiliary sampling budget as ARC; only the selection rule changes.

Condition	Cand.	Calls/turn	Selection rule	What it isolates	Reported metrics
HARNESS base	1	1	base action	base trajectory policy	solve, DIFF, Waste, Repeat, Blame
scheduled-`blame`	1	1	fixed diagnostic cadence	diagnosis frequency without candidate re-ranking	Blame vs. Waste/Repeat/DIFF
best-of-N	$3+3$	6	max $\hat{\pi}_t$	extra samples without the ARC utility	solve and process deltas vs. HARNESS
heuristic rerank	$3+3$	6	diagnose-first, local-edit, no-repeat rules	lightweight hand-coded trace preference	DIFF/Waste gains vs. ARC
ARC full	$3+3$	6	Eq. (4) + triggers	ARC utility and sparse intervention	joint DIFF/Waste/Repeat/Blame shift
$-\mathcal{U}_{\text{diag}}$	$3+3$	6	ARC without diagnostic value	same compute, no diagnostic term	Blame and downstream Waste
$-\Phi_{\text{spr}}$	$3+3$	6	ARC without spread penalty	same compute, no locality penalty	DIFF and Min-span
$-\Phi_{\text{loop}}$	$3+3$	6	ARC without loop penalty	same compute, no loop penalty	Waste and Repeat

Table D.8: **OpenAI comparison with matched proposal budgets.** Waste and Blame are means over completed runs; Edit is file precision over completed runs with edits. Err.@126 projects the observed error fraction to 126 tasks, rounded to the nearest integer. Cost is the mean cost per completed run, normalized within each model so ARC equals 1.00.

Model	Selection rule	Err.@126	Edit↑	Waste↓	Blame↑	Cost/ARC
gpt-4o-mini	base	0	0.33	4.60	1.90	0.19
gpt-4o-mini	scheduled-blame	0	0.44	4.90	4.20	0.16
gpt-4o-mini	best-of-N	0	0.23	4.10	1.50	1.16
gpt-4o-mini	heuristic	0	0.44	4.20	3.40	1.14
gpt-4o-mini	ARC	0	0.57	2.60	4.90	1.00
GPT-5.4-nano	base	0	0.67	9.00	1.40	0.21
GPT-5.4-nano	scheduled-blame	0	1.00	8.40	3.70	0.18
GPT-5.4-nano	best-of-N	0	0.43	10.50	0.60	1.29
GPT-5.4-nano	heuristic	0	0.45	5.80	4.10	1.30
GPT-5.4-nano	ARC	13	0.67	1.44	4.67	1.00
GPT-5.4-mini	base	0	0.75	4.40	2.30	0.23
GPT-5.4-mini	scheduled-blame	0	1.00	5.20	3.80	0.18
GPT-5.4-mini	best-of-N	0	0.50	6.90	1.40	1.30
GPT-5.4-mini	heuristic	0	0.57	3.20	4.00	1.32
GPT-5.4-mini	ARC	0	0.83	1.20	4.30	1.00
GPT-5.4	base	0	0.60	1.40	1.60	0.27
GPT-5.4	scheduled-blame	0	0.88	2.60	3.30	0.20
GPT-5.4	best-of-N	0	0.89	2.70	1.70	1.35
GPT-5.4	heuristic	28	0.70	1.14	2.43	1.54
GPT-5.4	ARC	0	1.00	0.80	4.70	1.00

Edit is the fraction of edited files that contain an injected fault; DIFF measures locality at the line level. Costs are averaged over completed trajectories, with failed attempts excluded from the cost totals and run counts. For GPT-5.4, the Cost/ARC values imply ARC/best-of-N cost $= 1/1.35 = 0.74$ and ARC/heuristic cost $= 1/1.54 = 0.65$.

Table D.9: **Matched compute on the 126-task manifest** (frozen Qwen2.5-Coder-14B; every selection rule receives the same three proposals plus three auxiliary generations, six calls per executed turn). DIFF is averaged over the edit-bearing trajectories counted in parentheses. See Section 6.3.

Selection rule	Solve	Stage	DIFF (n)	Waste↓	Repeat↓	Blame
Base (first proposal)	3/126	0.064	0.035 (43)	8.56	3.54	2.90
Best-of-N	3/126	0.056	0.012 (43)	8.90	3.18	2.86
Heuristic rerank	2/126	0.070	0.012 (41)	5.71	2.23	3.06
ARC	**10/126**	**0.121**	**0.167** (36)	**0.33**	**0.00**	**4.35**

Paired outcome split: 7 ARC-only, 0 base-only, 3 both, 116 neither; exact two-sided McNemar $p = 0.016$. All 504 trajectories complete with zero errors.

Table D.10: **Executed branch outcomes at 400 replayed states.** Each cell counts the states in which the selector chose a candidate of that type, out of the states where one was available. Waste is the share of wasteful selections; regret counts states where the selected candidate ranks below the executable oracle's choice.

Selector	Diagnosis	Improving edit	Stage advance	Task solve	Useful verify	Waste	Regret
Base-preferred	57/68	4/11	6/11	5/6	11/23	40.0%	153/400
Best-of-N	57/68	1/11	6/11	5/6	10/23	41.2%	147/400
Heuristic rerank	58/68	4/11	1/11	5/6	23/23	31.8%	102/400
ARC	59/68	5/11	0/11	5/6	23/23	34.2%	133/400
Executable oracle	68/68	11/11	11/11	6/6	20/23	35.8%	0/400

D.14 ALL PAIRED NUMBERS ON THE 126-TASK MANIFEST

Table D.11 consolidates all paired numbers on the 126-task manifest reported across the main text. External transfer numbers are reported in Table D.12 (GPT-5.4 across the three CLI benchmarks) and Table D.13 (cross-LLM transfer averaged across benchmarks).

Table D.11: **All paired numbers on the 126-task manifest.** All rows correspond to the same 126-task manifest under matched short budgets; this table consolidates the numbers reported in the main text and Figure 5 into one place. DIRECT and HARNESS rows use identical column definitions; shared metrics agree with Figure 5, which rounds some values for display.

Model	Cond.	n	Solve↑	DIFF↑	Rep.↓	Waste↓	Blame↑
Direct paired (Qwen3-32B, local inference)							
Qwen3-32B	DIRECT	126	21.4%	0.365	0.33	3.30	0.00
Qwen3-32B	DIRECT+ARC	126	34.1%	0.570	0.36	3.60	0.00
Harness paired (gpt-4o-mini)							
gpt-4o-mini	HARNESS	126	5.6%	0.648	2.10	6.18	2.48
gpt-4o-mini	HARNESS+ARC	126	6.3%	0.798	0.00	1.34	5.20
Same-model isolation (Qwen3-32B in the HARNESS *setting)*							
Qwen3-32B	HARNESS	126	24.6%	0.420	1.50	4.95	1.55
Qwen3-32B	HARNESS+ARC	126	37.3%	0.620	0.00	1.85	4.05
Strong-model harness transfer (fixed ARC *hyperparameters)*							
GPT-5.4	HARNESS	126	12.7%	0.27	1.12	4.34	1.28
GPT-5.4	HARNESS+ARC	126	20.6%	0.32	0.00	2.67	4.19
GPT-5.4-mini	HARNESS	126	9.5%	0.88	1.81	5.11	2.09
GPT-5.4-mini	HARNESS+ARC	126	11.1%	0.96	0.00	1.64	4.82
GPT-5.4-nano	HARNESS	126	1.6%	0.91	4.22	9.64	1.50
GPT-5.4-nano	HARNESS+ARC	126	1.6%	0.73	0.00	1.27	4.92

D.15 STRONG-MODEL CONTROLLER CHECK

Figure 5 tests whether ARC reduces waste and exact repeats and increases `blame` calls with stronger models. The main harness comparison used `gpt-4o-mini`; we reuse its controller without retuning.

Figure D.7 reports three analyses for the stronger models. Panel (a) compares pass rate, wasted turns, exact repeats, and `blame` counts across models and conditions. Within each model, HARNESS and HARNESS+ARC are paired; HARNESS+ARC reduces waste and repeats and increases diagnosis frequency on the 126-task set. Panel (b) compares scaled improvements in solve, waste, repeats, and diagnosis for each model. Panel (c) plots diagnosis frequency against wasted turns. Arrows connect the paired conditions for each model, and marker area encodes API cost per task. These results support transfer of the process effects, while solve rates differ across the three bases.

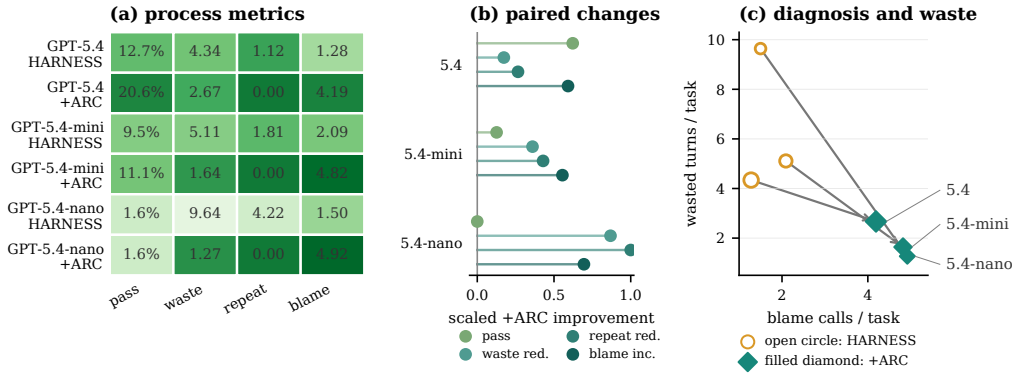


Figure D.7: **Strong-model controller check across the GPT-5.4 family.** Panel (a): process matrix of pass rate, waste, repeats, and `blame` per strong-model condition. Panel (b): scaled ARC improvements by model. Panel (c): diagnosis frequency and wasted turns, with arrows connecting paired conditions and marker area encoding API cost per task. The waste and repeat effects of ARC carry to stronger base models; solve rate is a separate axis.

D.16 PER-LLM BY BENCHMARK TRANSFER BREAKDOWN

Table D.12 gives the GPT-5.4 benchmark-by-benchmark transfer behind Figure 5(b). Table D.13 then expands the cross-LLM average in Figure 5(c) into the underlying 36 cells: three model families (gpt-4o-mini, Llama-3.3-70B-Instruct, Claude-Opus-4.6) by three benchmarks by four metrics. Native outcomes and all three process proxies improve for every model and benchmark listed.

We also evaluated 12 naturally occurring bugs requiring edits across two to five files. Tasks were selected before observing outcomes and used the same CLI protocol without annotated fault regions. Base solved 3 of 12 tasks and ARC solved 4; mean wasted turns fell from 5.08 to 1.67, repeats from 1.58 to 0.17, and diagnostic calls rose from 2.3 to 4.8.

Table D.12: **GPT-5.4 external transfer by benchmark.** Per-benchmark expansion of Figure 5(b). Each cell reports vanilla → +ARC under the fixed controller; positive deltas in Outcome / Focus / Scope / Stability indicate improvement.

Benchmark	Outcome↑	Focus↑	Scope↑	Stability↑
TerminalBench	58.4 → 62.9	0.52 → 0.66	0.61 → 0.71	0.55 → 0.78
AgencyBench	48.4 → 53.6	0.49 → 0.62	0.58 → 0.67	0.52 → 0.74
LongCLI Bench	19.0 → 23.5	0.41 → 0.55	0.49 → 0.61	0.43 → 0.68

Table D.13: **Cross-LLM × per-benchmark transfer.** Each cell reports vanilla → +ARC under the fixed controller; positive deltas in Outcome / Focus / Scope / Stability indicate improvement.

Base	Benchmark	Outcome↑	Focus↑	Scope↑	Stability↑
gpt-4o-mini	TerminalBench	$33.4 \rightarrow 36.1$	$0.46 \rightarrow 0.59$	$0.54 \rightarrow 0.64$	$0.50 \rightarrow 0.69$
gpt-4o-mini	AgencyBench	$26.8 \rightarrow 29.4$	$0.43 \rightarrow 0.55$	$0.51 \rightarrow 0.60$	$0.48 \rightarrow 0.66$
gpt-4o-mini	LongCLI Bench	$13.3 \rightarrow 15.8$	$0.40 \rightarrow 0.53$	$0.47 \rightarrow 0.57$	$0.46 \rightarrow 0.63$
Llama-3.3-70B	TerminalBench	$44.2 \rightarrow 47.9$	$0.49 \rightarrow 0.62$	$0.58 \rightarrow 0.67$	$0.53 \rightarrow 0.72$
Llama-3.3-70B	AgencyBench	$34.4 \rightarrow 38.2$	$0.46 \rightarrow 0.59$	$0.55 \rightarrow 0.64$	$0.50 \rightarrow 0.69$
Llama-3.3-70B	LongCLI Bench	$19.5 \rightarrow 23.5$	$0.43 \rightarrow 0.56$	$0.51 \rightarrow 0.60$	$0.46 \rightarrow 0.65$
Claude-Opus-4.6	TerminalBench	$63.2 \rightarrow 66.7$	$0.57 \rightarrow 0.71$	$0.66 \rightarrow 0.75$	$0.61 \rightarrow 0.81$
Claude-Opus-4.6	AgencyBench	$52.7 \rightarrow 56.5$	$0.54 \rightarrow 0.68$	$0.63 \rightarrow 0.72$	$0.58 \rightarrow 0.78$
Claude-Opus-4.6	LongCLI Bench	$39.0 \rightarrow 42.4$	$0.50 \rightarrow 0.64$	$0.59 \rightarrow 0.69$	$0.55 \rightarrow 0.74$

Three researchers who were not authors independently reviewed 30 pairs of trajectories with the same task, model, and budget. The sample included 10 pairs from each external benchmark, with 16 pairs in which both conditions passed and 14 in which both failed. Each reviewer had 1 to 3 years of software engineering experience and had not participated in the project or developed its proxy metrics or review rubric. The sampling seed was fixed in advance, and A/B assignment was randomized separately for each pair and reviewer. Proxy definitions and values, utility scores and triggers, condition and model labels, and the authors' earlier audit materials were withheld.

The rubric and analysis plan were fixed before reviewer recruitment. Reviewers answered three questions, each with A/B/tie. The diagnosis question was: *Which trajectory more clearly identifies where and why the task is failing before making changes?* The edit question was: *Which trajectory makes changes better supported by the evidence it has gathered, with fewer unnecessary edits?* The overall quality question was: *Taking everything into account, which repair would you rather inherit as the basis for review and further maintenance?* They also scored evidence grounding, change discipline, process stability, verification quality, and suitability for further work on a five-point scale from 1 (very weak) to 5 (very strong). We average three ratings per pair; 95% intervals use pair bootstrap.

For the overall quality question, the 90 judgments were 64 for ARC, 11 for the base, and 15 ties. A pair was assigned a condition preference only when at least two of its three reviewers selected that condition; all other pairs were treated as ties. This gave 22 ARC preferences, 3 base preferences, and 5 ties, with Fleiss' $\kappa = 0.54$; a sign test over pairs gives $p = 1.6 \times 10^{-4}$. All five rating dimensions favored ARC.

Table D.14: **Independent non-author review of 30 matched trajectory pairs.** Mean ratings on a five-point scale, averaged over three reviewers per pair. Changes are ARC minus base, with 95% confidence intervals from paired bootstrap over pairs.

Dimension	Base	ARC	Change [95% CI]
Evidence grounding	3.1	3.9	$+0.8$ $[+0.5, +1.1]$
Change discipline	3.0	3.8	$+0.8$ $[+0.4, +1.1]$
Process stability	3.2	4.0	$+0.8$ $[+0.5, +1.2]$
Verification quality	3.3	3.9	$+0.6$ $[+0.3, +0.9]$
Suitability for further work	3.1	4.0	$+0.9$ $[+0.6, +1.2]$

E CASE STUDIES AND FORENSIC TIMELINES

Each case compares where agents edit, when they diagnose and test, and whether the repair succeeds on the same task. Figure 7 gives candidate action probabilities for the first case.

E.1 DIAGNOSIS-FIRST REPAIR CONVERTS A FAIL INTO A PASS

Benchmark: TRACEBENCH-CLI **Task:** `tb_hard_00045` **Bug:** typo in the package-initialization file

Protocol: 126 paired run, `gpt-4o-mini`, short budget (20 turns).

HARNESS trajectory (20 turns, fail). The agent spends 16 turns in Stage 0 and repeatedly edits an unrelated setup file before eventually touching the correct resource. Only one blame call occurs, and it happens at turn 20, after the budget has effectively been exhausted.

HARNESS+ARC trajectory (15 turns, pass). The controller permits a single targeted edit at turn 3 and requests diagnosis after each stage advance. Diagnosis precedes the localized edit, and verification follows immediately.

E.2 DIFFUSE EDITS VERSUS SYSTEMATIC DIAGNOSIS IN FAILED REPAIRS

Benchmark: TRACEBENCH-CLI **Task:** `tb_hard_00042` **Bug:** environment fault (Makefile path)

Protocol: 126 paired run, `gpt-4o-mini`, short budget (20 turns).

HARNESS trajectory (20 turns, fail). The agent emits seven edits across four different files, makes zero blame calls, and repeatedly alternates between speculative edits and failed tests. It never establishes which file is responsible.

HARNESS+ARC trajectory (20 turns, fail). The controller redirects each speculative edit into a diagnostic action. The resulting trajectory makes nine blame calls and no edits, implicating the build configuration early in the run. Both trajectories fail.

E.3 BOTH CONDITIONS PASS, BUT TRAJECTORY QUALITY DIFFERS

Benchmark: TRACEBENCH-CLI **Task:** `tb_hard_00101` **Bug:** logic error in `task.py`

Protocol: 126 paired run, `gpt-4o-mini`, short budget (20 turns).

HARNESS trajectory (8 turns, pass). The agent solves the task quickly with one edit, but makes no blame calls and leaves no explicit diagnosis trail.

HARNESS+ARC trajectory (14 turns, pass). The controller inserts a diagnosis before editing and multiple verification steps after the fix. Both conditions pass. The ARC trajectory makes the link between observations and edits explicit.

E.4 MATCHED FORENSIC TIMELINE

Figure E.1 compares ARC and base runs of `tb_hard_00045` under the same model and budget.

Panel (a) shows the HARNESS run: the x-axis is turn index and the y-axis is stage progress. The agent spends sixteen turns on an unrelated setup file before reaching the faulty file. Its single `blame` call occurs at turn 20, at the end of the budget. Panel (b) shows the HARNESS+ARC run on the same task: a targeted edit at turn 3, followed by diagnosis and verification. The edits remain in the file containing the injected fault. Panel (c) compares touched files with the known faulty file. Under HARNESS, most edits target the setup file; under HARNESS+ARC, edits and `blame` calls concentrate on the package initializer. The timeline shows how the two runs differ in edit targeting and the timing.

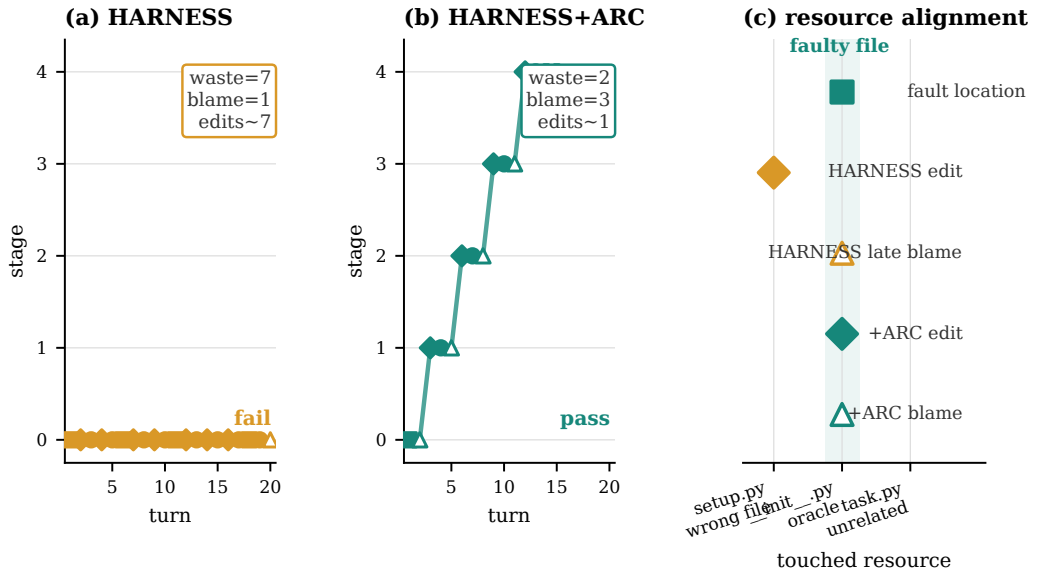


Figure E.1: **Matched forensic timeline for the case study.** Panels (a) and (b) show HARNESS and HARNESS+ARC stage progress at each turn under the same budget. Panel (c) compares touched files with the known faulty file. HARNESS+ARC concentrates edits and diagnostic calls on the package initializer; HARNESS makes most edits in the wrong file.

E.5 SCOPE AND LIMITATIONS

Section 6 combines controlled repair tasks with transfer experiments. On external CLI benchmarks, focus, scope, and stability from Appendix A.5 measure process behavior alongside native outcomes. Diff precision measures locality relative to a recorded repair span, and the complete hidden test suite determines success. The passing patch audit uses deletion and replay to identify necessary changes beyond that span. Branch replay measures candidate availability and selection accuracy.