EPISTEMIC DEBT IN CODING AGENTS: REPAYING UNSUPPORTED ASSUMPTIONS IN REPOSITORY-LEVEL SOFTWARE ENGINEERING

Anonymous authors
Paper under double-blind review

ABSTRACT

Repository-level coding agents act on provisional judgments while their evidence about a codebase is still incomplete. A wrong assumption about a call chain, a specification, or a test can silently become the premise of later edits, tests, and memory; every dependent action makes correcting it costlier. We call this accumulating exposure *epistemic debt*: the expected downstream correction cost of deferring verification of an action-dependent repository assumption. Existing agents verify reactively after a failure or indiscriminately before every write; neither selects which assumption is worth checking. We treat verification as a budgeted selection problem: a dynamic assumption graph links each falsifiable assumption to its evidence and dependent actions, and a budgeted controller runs the assumption–check pair whose expected avoided rework exceeds its cost. A paired counterfactual protocol measures that value by branching from the same checkpoint with and without a targeted check. On a 300-instance subset of SWE-bench (three seeds), the debt-aware agent raises resolution from 31.9% to 48.8% with fewer agent steps and lower dollar cost. The gain replicates across three agent scaffolds and two backbones. The exposure-aware heuristic already provides a substantial improvement, while a learned scorer adds a further 4.5 percentage points on the full benchmark as an exploratory in-sample estimate; its held-out end-to-end transfer remains a limitation.

1 INTRODUCTION

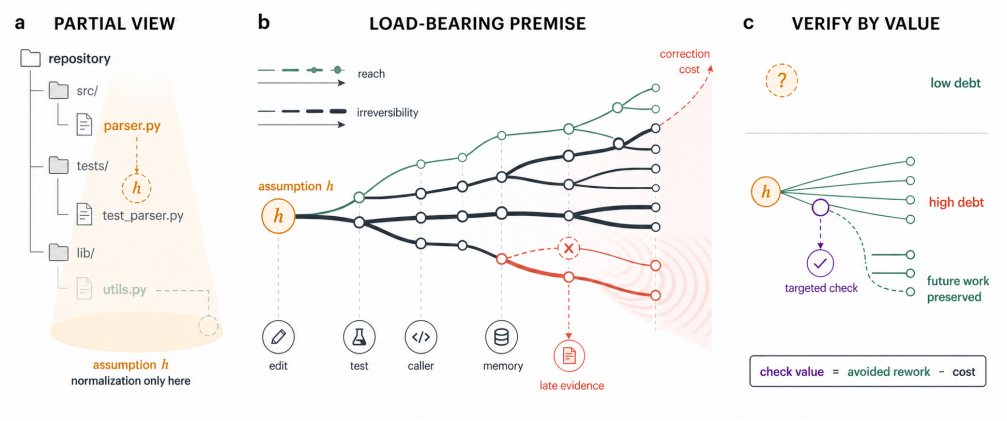

Figure 1: Epistemic debt in a repository agent. (a) With only a partial view, the agent forms a provisional assumption ("normalization lives only in `parser.py`") and acts on it. (b) As edits, tests, and memory come to depend on that premise, its correction cost compounds; late evidence forces rework across the whole dependent branch. (c) Verification is worth paying for exactly when it can redirect costly downstream decisions: a targeted check whose avoided rework exceeds its cost.

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.

Coding agents resolve issues in real software repositories. They search thousands of files, inspect definitions and call sites, modify several components, run tools and tests, and revise their plans over a long interaction (Jimenez et al., 2024; Yang et al., 2024; Xia et al., 2024). The agent often has to decide while its evidence about the repository is still incomplete. It forms provisional judgments about where a bug originates, which behavior the issue requests, or whether a test is discriminative. Such judgments are often necessary for progress.

A provisional judgment can silently become a durable premise. Suppose an agent sees a failing parser test and assumes that normalization occurs only in the parser. It edits the parser, writes a regression test, summarizes the diagnosis into memory, and changes a caller to accommodate the new behavior. A later search reveals a second normalization layer in a shared utility. What began as one unsupported claim now affects several files, a test, and the working context of the agent. As dependent actions accumulate, an unsupported assumption becomes more expensive to correct and can stay hidden until substantial work has been built on top of it. Recent trajectory analysis supports the importance of this pattern: in 1,184 failed coding-agent runs, epistemic errors are the largest root-cause category, and false premises alone account for 30.7% of decisive errors (Zhao et al., 2026).

We call this accumulating exposure *epistemic debt*: the expected downstream correction cost of deferring verification of an action-dependent repository assumption. We focus on *repository assumptions*: short, falsifiable propositions about a codebase that at least one observable action depends on. This definition separates epistemic debt from model uncertainty: a claim may be uncertain but harmless if no action depends on it, while a moderately confident claim can carry high debt when it supports an invasive, difficult-to-reverse patch.

Uncertainty alone does not determine debt: the probability that an assumption is wrong, the reach of the actions that depend on it, the irreversibility of those actions, and the price of the check that could settle it are distinct quantities. A system that verifies whenever it is uncertain, or verifies everything, or waits for a failure signal, solves a different problem.

Existing work addresses parts of this problem, but not as a budgeted selection problem. Reflection-based repair reacts after a test failure, when the wrong assumption has already propagated into dependent edits (Shinn et al., 2023; Huang et al., 2024). Unconditional grounding gathers context before actions, but it can incur substantial or linearly growing verification cost and does not select which assumption to audit (Jiang et al., 2026; Yuan et al., 2026). Adaptive-compute methods learn whether extra reasoning is useful at a state, but do not choose a concrete assumption or check (Li et al., 2026). We treat repository verification as a selection problem: which assumption–check pair to run, under a budget, so that the downstream rework avoided exceeds the cost of the check. The novelty claim is therefore not that verification helps in the abstract: it is the coupling of a repository-dependency exposure graph, a counterfactual avoided-rework estimand, and a budgeted assumption–check decision unit. If verification alone were sufficient, the verify-every-write policy should match the targeted policy at an equal or larger budget; Q1 and Q4 test this alternative directly.

We introduce a debt-aware coding agent organized around two components. A *dynamic assumption graph* records falsifiable repository assumptions, observed evidence, code entities, and agent actions, with typed edges that distinguish evidential support from downstream decision dependence. A *repayment controller* ranks assumption–check pairs (h, v) by their expected avoided rework and selects checks under a shared budget. A paired counterfactual protocol measures the value of verification by branching from the same trajectory checkpoint with and without a targeted check. The exposure-aware objective and decision unit are the central contribution; the learned scorer is a replaceable implementation that refines this ranking rather than the claimed source of generality.

We make three contributions:

- We define *epistemic debt* as the counterfactual, expected downstream correction cost of deferring verification of an action-dependent repository assumption, separate from uncertainty, factual error, and generic trajectory complexity.
- We build a dynamic assumption graph and a budgeted repayment policy whose action is a concrete assumption–check pair and whose objective is expected avoided rework.
- We measure verification value with paired counterfactuals and separate the system-level effect from the scorer effect: the exposure-aware heuristic reaches 44.3% versus 31.9% without verification, while learned scoring adds 4.5 percentage points on the full benchmark

as an exploratory in-sample estimate. The held-out SymPy/xarray end-to-end gain is not significant, so scorer transfer is a stated limitation rather than a claimed success.

The contribution is a *within-scaffold causal estimate*: we hold the scaffold, backbone, and tool interface fixed and vary only the verification policy, so the reported gains isolate the mechanism rather than a stronger system. Public leaderboard scores on SWE-bench mix scaffolds, backbones, and budgets and are listed for reference in App. C.7, not as comparison points.

2 Related Work

2.1 Coding Agents, Failure Propagation, and Patch Rework

Repository-level coding agents search, edit, and test across large codebases, and the field now has mature benchmarks for end-to-end performance (Jimenez et al., 2024; Yang et al., 2024; Xia et al., 2024). A complementary line dissects *why* these agents fail. Trajectory-level analysis finds that false premises and unsupported claims dominate long-horizon failures, and that decisive errors often form well before their consequences become observable (Zhao et al., 2026). On the repair side, Mathai et al. (2026) minimize completed patches by removing functionally unnecessary edits. Graph-guided navigation represents codebases as dependency graphs to help agents localize the right entities (Chen et al., 2025). Our focus is the runtime decision of which assumption to verify before it propagates, and which check avoids the most downstream rework.

2.2 Explicit Beliefs, Assumptions, and Grounding

Several methods make the beliefs or assumptions of an agent explicit and subject them to external evidence. Yuan et al. (2026) audit unsupported internal beliefs before action or memory commitment; their unit is a belief-state audit and their objective is reasoning faithfulness, not repository dependency or marginal avoided rework. Jiang et al. (2026) proactively gather missing environmental context before actions. Our verify-every-write baseline is an ACCORD-style adaptation to repository editing, rather than a reproduction of ACCORD itself, and it does not rank assumption–check pairs by counterfactual rework savings. Wu et al. (2026) extract implicit assumptions from generated code and trace them for developer inspection. Their target is assumption extraction and localized regeneration at code-generation time, whereas ours is online runtime trajectory selection under a finite budget. Structured belief states with process-level rewards (Tang et al., 2026), environment probing to realign memory (Yin & Du, 2026), and belief auditing in games (Gao et al., 2026) each verify at a different unit (belief, experience, or game state). What differs is the estimand and decision unit: a falsifiable repository proposition whose correction cost grows through *code* dependencies, and the marginal downstream rework avoided by verifying one concrete assumption–check pair rather than belief consistency alone. A compact nearest-neighbor comparison is provided in App. A.1. This also makes the claim falsifiable: a generic verifier should be competitive with targeted selection under the same budget, whereas our verify-every-write baseline is both less effective and more expensive. The contribution is thus a problem formulation with a measurable prediction, not a component-inventory claim that any verifier is novel.

2.3 Selective Verification, Uncertainty, and Value of Information

Adaptive-compute methods learn whether extra reasoning is useful at a state. Li et al. (2026) show that uncertainty does not have a stable relationship with the value of additional computation and learn a state-level intervention gate from counterfactual exploration; our method instead operates at the assumption–check level and defines utility through avoided repository rework. This is the classical value-of-information framing (Howard, 1966): a repository check is an information-gathering action whose value is the rework it can prevent. Verification is not a silver bullet: software verifiers are incomplete proxies for intended behavior, so evidence sufficiency, claim truth, and impact must remain separate quantities (Wang et al., 2026).

Boundary of the claim. The nearest methods differ along three axes that are easy to conflate: the object being verified (belief, missing context, generated-code assumption, or repository assumption), the granularity of intervention (state/action versus one assumption–check pair), and the utility being

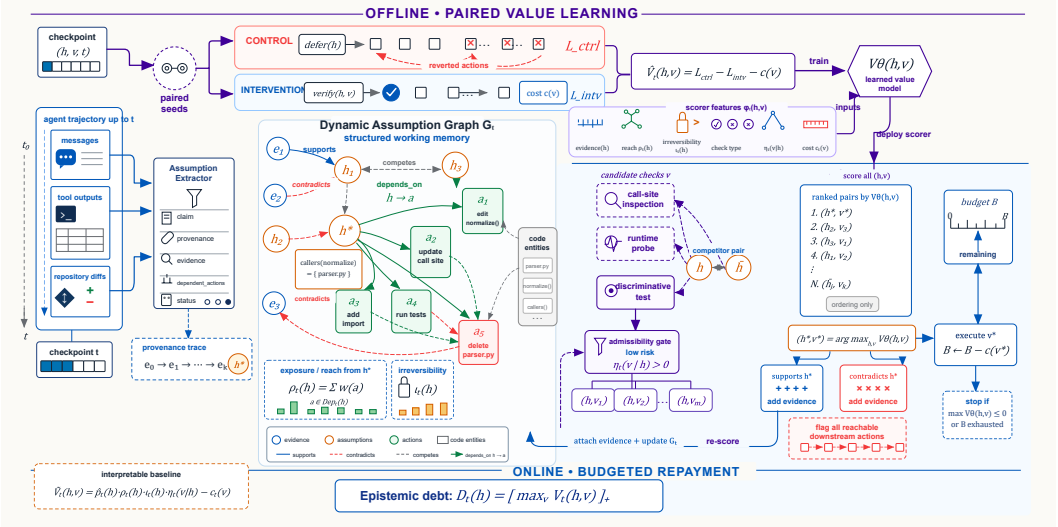

Figure 2: Overview of epistemic-debt learning and repayment. Offline, paired continuation seeds supervise $V_\theta(h, v)$ with realized avoided rework. Online, visible trajectory evidence is extracted into the dynamic assumption graph G_t; candidate checks are ranked, the highest positive pair executed under a budget, and its outcome attached as new evidence before re-scoring. The `depends_on` direction is fixed as $h \to a$; the dashed factorization at lower left is an interpretable baseline for comparison.

optimized (faithfulness, grounding, localization, or avoided downstream rework). Our claim is specifically the joint effect of the latter two axes under a shared budget; we do not claim novelty for self-auditing, context gathering, or assumption extraction in isolation.

On the term *epistemic debt*. Prior work uses it to describe knowledge deficits in human developers under AI-assisted ("vibe") coding, where novices outsource understanding they later need to maintain code (Sankaranarayanan, 2026). That line measures a *human* cognitive deficit; we define epistemic debt as a *counterfactual cost within an agent trajectory*. Similarly, technical debt in ML systems (Sculley et al., 2015) and agentic technical debt (Hydari et al., 2026) concern architectural, memory, and orchestration maintenance costs of the *system*, not the deferred-verification cost of a specific repository assumption.

3 Method

The method acts on concrete *assumption–check pairs* (h, v). At each checkpoint, a dynamic assumption graph makes repository assumptions explicit; candidate checks can settle each assumption; a learned value model ranks pairs; and a budgeted controller executes the check that avoids the most downstream rework. Fig. 2 summarizes the offline learning signal and the online repayment loop; the notation used throughout this section is listed in Table 3 (App. B.1).

3.1 Problem Formulation and Observability

We model a coding agent as a sequence of actions $a_1, \ldots, a_T$ over a repository. At checkpoint t, the visible trajectory τ_t (messages, tool outputs, and repository diffs produced so far) is the evidence the agent conditions on, and the next action a_{t+1} is about to be chosen. A *repository assumption* h is a proposition that satisfies three conditions: (i) it appears in the visible trajectory τ_t; (ii) it is a short, falsifiable proposition about the repository; and (iii) at least one observable action depends on it. The actions that presuppose h form its dependent set,

$$\mathrm{Dep}_t(h) = \{a : a \text{ is an action in } \tau_t \text{ that presupposes } h\}, \tag{1}$$

so condition (iii) reads $\mathrm{Dep}_t(h) \neq \emptyset$. We distinguish *stated* assumptions, which satisfy (i) directly, from *behavior-inferred* assumptions, which we read off from code changes when no explicit statement

exists. Behavior-inferred assumptions play a secondary role: the primary experiments use stated assumptions, and we report how many trajectories fail to recover a clear assumption.

Assumptions are produced by an extractor, $\mathcal{H}_t = \textsc{Extract}(\tau_t)$, a prompt-based LLM extractor on the same backbone as the agent (deepseek-chat) that reads the visible context and emits structured assumption records with provenance and dependency edges (schema in App. B.3). The extractor runs after each agent step; it is *not* trained; its prompt was designed against the manual annotation guide and validated on the annotated pilot trajectories (Sec. 3.4), which also provide the precision/recall and agreement estimates. Duplicate or synonymous assumptions are merged by normalized-claim matching that keeps the earliest provenance, and conflicting assumptions are kept distinct and linked by competing edges.

We evaluate extraction with precision, recall, and annotator agreement (Sec. 4.4); the full schema and prompts are in the appendix.

3.2 Dynamic Assumption Graph and Check Generation

The assumption graph $G_t = (N_t, E_t)$ records what the agent has assumed, observed, and done. Nodes are typed and partitioned into four sets, and edges carry a type from a fixed vocabulary,

$$G_t = (N_t, E_t), \quad N_t = N_t^h \cup N_t^e \cup N_t^c \cup N_t^a, \quad E_t \subseteq N_t \times N_t \times \mathcal{T}, \tag{2}$$

where N_t^h, N_t^e, N_t^c, N_t^a are assumption, evidence, code-entity, and action nodes, and $\mathcal{T} = \{\texttt{supports}, \texttt{contradicts}, \texttt{refines}, \texttt{competes}, \texttt{depends_on}\}$. Edges behave as follows: `supports` and `contradicts` connect evidence to an assumption; `refines` and `competes` relate assumptions to each other; and `depends_on` connects an assumption h to each action a that presupposes it, with the direction fixed as $h \to a$, so that forward traversal from h reaches exactly its dependent actions.

The *exposure* of h, denoted $\rho_t(h)$, is a weighted sum of the cost weights of its dependent actions,

$$\rho_t(h) \;=\; \sum_{a \in \mathrm{Dep}_t(h)} w(a), \tag{3}$$

where $w(a)$ is a cost weight for action a; in this work it is instantiated as normalized agent steps (one tool call $= 1$, an edit scaled by its diff size), with the USD conversion specified in App. B.9. Exposure, not raw confidence, is the primary structural signal for ranking, and it grows monotonically as the dependent set grows (Lemma 1, App. A). Weighting rather than counting keeps a single large edit from dominating the ranking; a code entity and the action that edits it are not counted as two separate exposures, and we control for trajectory length in evaluation.

The graph is updated incrementally: each executed check attaches one evidence node with a `supports` or `contradicts` edge,

$$G_{t+1} = \textsc{Update}(G_t, o), \tag{4}$$

where o is the check outcome; reach is then recomputed only for the affected assumption.

For a live assumption h, we generate a small set of candidate checks $\mathcal{V}(h)$ that can produce repository or execution evidence: (i) *call-site / reference inspection*, enumerating call sites, overrides, or imports; (ii) *runtime probe*, inspecting a type, state, or configuration value; and (iii) *discriminative test*, a minimal test constructed to separate h from its strongest competitor. Branch comparison is deferred to the appendix. A check is admissible only if it is low-risk and *discriminative*: its outcome o_v should differ in distribution between the hypothesis that h is true and the hypothesis that its strongest competitor h' is true, that is, $\mathbb{P}(o_v \mid h) \neq \mathbb{P}(o_v \mid h')$. We estimate discriminativeness $\eta(v \mid h) \in [0, 1]$ with a static rule keyed on check type (discriminative tests score highest, runtime probes next, then call-site inspection); a learned estimator may replace it; when no explicit competitor exists, we generate the negation of h or the nearest alternative explanation as the competitor, and if no check can produce a different prediction under the two hypotheses, it is dropped from the candidate set.

3.3 Counterfactual Verification Value

A rework loss is the primary outcome. Rather than summing heterogeneous raw quantities (reverted lines, wrong-file edits, failed tests, tokens), we define the rework attributed to a set of invalidated

actions as the sum of their cost weights, $L^{\text{rework}}_{>t} = \sum_{a \in \text{InvalidatedActions}_{>t}} w(a)$, where $w(a)$ is a unified action cost (e.g., agent steps or a normalized tool cost). Other metrics (reverted or superseded edits, wrong-file changes, failed tests, tokens, tool calls, wall-clock, and task resolution) are reported separately and never combined into a single score without an explicit weight source.

Unit convention. Count-based rework metrics are reported as counts. The value model instead operates in dollars: $L_{>t}$ in Eqs. 5–7 is the USD API cost of the invalidated actions and $c_t(v)$ the USD cost of the check, so the subtraction in Eq. 5 is well defined (full conversion table in App. B.9).

The repayment controller ranks assumption–check pairs (h, v) by their expected *counterfactual verification value*, which we define exactly as

$$V_t(h, v) = \mathbb{E}[L_{>t} \mid \text{defer}(h)] - (c_t(v) + \mathbb{E}[L_{>t} \mid \text{verify}(h, v)]), \tag{5}$$

so that the debt is $\mathcal{D}_t(h) = [\max_v V_t(h, v)]_+$, which is non-decreasing in the dependent set of h while the error probability and the available check set are unchanged (Prop. 2, App. A); new evidence can of course update $p_t(h)$ and reset the debt.

The full method estimates $V_t(h, v)$ directly with a learned scorer V_θ trained on paired counterfactual outcomes (Sec. 3.4). As an interpretable baseline we also define a heuristic approximation,

$$\widetilde{V}_t(h, v) \;=\; \widehat{p}_t(h)\; \rho_t(h)\; \iota_t(h)\; \eta_t(v \mid h) \;-\; c_t(v), \tag{6}$$

where $\widehat{p}_t(h)$ is an error-probability estimate read from the evidence subgraph of h (an LLM weighs the `supports` and `contradicts` evidence attached to h), $\iota_t(h)$ is the irreversibility of the dependent edits (edits to shared callers or to files with existing tests are harder to undo than isolated ones), and $\eta_t(v \mid h)$ is the discriminativeness of the check. Under a perfect-check assumption, the value of a check decomposes exactly into error probability and exposure (Prop. 1, App. A); the heuristic of Eq. 6 estimates this decomposition, with $\iota_t(h)$ and $\eta_t(v \mid h)$ discounting the proposition's idealizations (not every dependent edit is equally hard to undo, and real checks are not perfect), while the learned V_θ can capture departures from it. The learned scorer does not assume these factors combine multiplicatively, and can capture interactions such as a verification that disturbs the agent even when h is correct, or a check that returns ambiguous evidence.

3.4 Paired Training Signal and Budgeted Repayment

The training signal for V_θ comes from paired counterfactuals. From a trajectory checkpoint where h is live, we fork a *control* branch that continues unchanged and an *intervention* branch that executes a selected check v and injects its objective outcome before continuing. If the intervention loss $L^{\text{intv}}_{>t}$ excludes the cost of v itself, the realized value is

$$\widehat{V}_t(h, v) \;=\; L^{\text{ctrl}}_{>t} - L^{\text{intv}}_{>t} - c_t(v), \tag{7}$$

where the loss of each branch is the rework defined above. Each checkpoint–check pair is rolled out with several continuation seeds, with control and intervention sharing paired seeds, and we report the mean and a bootstrap confidence interval; no individual branch is treated as a causal claim. To avoid selection bias in training, data collection mixes random-selected, uncertainty-selected, and heuristic-selected pairs, and train/test splits are by repository to prevent leakage.

The controller is a simple, reproducible sequential policy: (1) generate all currently available pairs (h, v); (2) predict $V_\theta(h, v)$ for each; (3) select the pair with the largest positive value; (4) if the largest value is not positive, do not verify; (5) execute the check; (6) update the graph with the outcome, flagging live actions reachable from a refuted h; (7) re-score all pairs; (8) stop when the budget is exhausted. Under a unit intervention budget, selecting the pair with the largest positive estimated value is consistent with the estimated objective (Prop. 3, App. A); under larger budgets we apply sequential re-ranking because each observation can change the value of the remaining checks, and Prop. 4 (App. A) bounds the single-round loss of a greedy choice by 2ϵ under an ϵ-accurate scorer. No constant-factor guarantee applies to the sequential process: checks are interdependent, the objective is not monotone submodular, and the remaining budget depends on earlier outcomes, so the multi-round controller is validated empirically. Algorithm 1 in App. B.2 gives the complete pseudocode.

Cost. The controller adds one extractor call per agent step and, per round, $O(|\mathcal{P}|)$ scorer evaluations over the candidate pairs ($|\mathcal{P}| \leq 8$ here, at most B rounds), so extraction is the only LLM query beyond the agent's own calls; the per-instance dollar cost of the debt-aware agent therefore stays at or below the no-verification baseline (Table 1).

4 EXPERIMENTS

We test whether debt-aware selection reduces rework under a matched budget (Q1), which value component drives the gain (Q2), whether assumptions are recoverable and checks change downstream edits (Q3), where the budget goes (Q4), and whether reach predicts repayment value (Q5). Full prompts, hyperparameters, annotation rules, and per-repository statistics are in the appendix.

4.1 EXPERIMENTAL SETUP

Setting. We use a 300-instance subset of Jimenez et al. (2024) spanning twelve repositories. Every main configuration uses mini-swe-agent 2.4.6 with deepseek-chat, the same tools and stopping conditions, and three fixed seeds; paired forks use three matched continuation seeds. A separate 10-instance pilot fixed the annotation and fork protocols before the full run (App. C.12).

Metrics. The primary outcome is downstream rework, reported as wasted actions and reverted diff lines. We separately report task resolution, agent steps, and dollar cost; unit definitions are in App. B.9.

Baselines. Under the same budget we compare no verification, a random live pair, uncertainty ranking, an ACCORD-style verify-every-write policy (Jiang et al., 2026), post-failure reflection (Shinn et al., 2023), a state-level adaptive-compute gate (Li et al., 2026), the fixed-factor heuristic of Eq. 6, and our learned controller. Two controls isolate selection from execution: selection-only (random live pair with debt-aware's check count) and equal-cost (random live pair until debt-aware's total check spend). We treat verify-every-write as the executable proxy for a SAVeR-style per-step belief audit (Yuan et al., 2026), which would add extractor-level cost without producing discriminating evidence (App. C.12). Truth and value oracles upper-bound assumption recovery and pair selection; App. B.8 specifies every execution rule.

Budget and statistics. Every policy receives verification budget $B = 20$ in normalized check-cost units (call-site 1.0, runtime probe 1.5, test 2.0); ordinary agent steps are not charged to B. Realized consumption is reported in App. C.4. Resolution comparisons use 2,000 instance-block bootstrap resamples that pool the three seeds of each sampled instance; step counts use Cohen's d, and scorer quality uses held-out repositories (full protocol in App. 4).

Censoring and comparability. The main claim is a within-scaffold comparison; truncation sensitivity, extractor censoring rates, and the transfer-label definitions are in Apps. C.13, C.1, and C.7.

4.2 DOES DEBT-AWARE SELECTION REDUCE REWORK UNDER A MATCHED BUDGET? (Q1)

Debt-aware selection resolves 48.8% of instances versus 31.9% without verification, a $+16.9$ pp gain (block-bootstrap 95% CI $[+12.7, +21.0]$), while steps fall from 27.2 to 22.4 ($d = -0.90$) and cost from $\$0.072$ to $\$0.059$ (Table 1). Verify-every-write is both weaker (39.9%) and costlier ($\$0.087$), while uncertainty ranking and post-failure reflection are not significant improvements. Random selection does not buy the gain at equal cost: selection-only (random live pair, matched check count) resolves 35.6% (CI $[31.9, 39.6]$) and equal-cost (random checks, matched spend) 34.8% (CI $[31.2, 38.6]$), both below the heuristic's 44.3%—the benefit comes from choosing the right pair, not from executing more or different checks. As an exploratory in-sample estimate, the learned scorer adds 4.5 pp (CI $[+3.0, +6.0]$). The value oracle reaches 54.9%, leaving 6.1 pp of selection headroom; the truth oracle reaches 50.9% and is not significantly better than our controller.

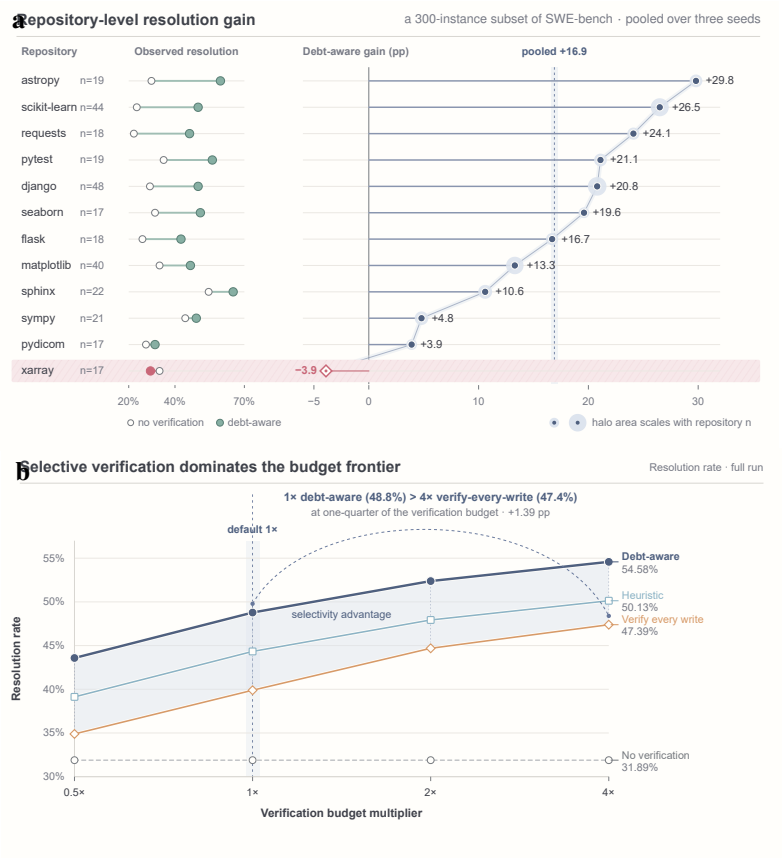

Figure 3: Cross-repository generality and budget robustness. **a**, Per-repository resolution gain over no verification; xarray (-3.9 pp) is the sole exception. **b**, Resolution as the verification budget varies; debt-aware selection at $1\times$ (48.8%) exceeds verify-every-write at $4\times$ (47.4%). Results pool 300 instances and three seeds.

Table 1: Results on a 300-instance subset of SWE-bench (three seeds). CIs use the instance-block bootstrap; steps and costs are per instance.

Configuration	Resolved ↑	95% CI	Steps ↓	Mean cost ($) ↓
No verification	31.9%	28.1–35.7	27.2	0.072
Random pair	33.9%	30.1–37.9	27.2	0.072
Selection-only (random, matched checks)	35.6%	31.9–39.6	24.3	0.062
Equal-cost (random checks, matched spend)	34.8%	31.2–38.6	24.9	0.059
Uncertainty rank.	35.0%	31.3–39.0	27.5	0.072
Post-fail. reflection	34.4%	30.8–38.3	29.4	0.077
State-level gate	34.7%	31.0–38.7	27.1	0.071
Verify-every-write	39.9%	35.8–43.8	33.2	0.087
Heuristic debt score	44.3%	40.2–48.3	23.9	0.063
Debt-aware (ours)	**48.8%**	44.8–52.9	**22.4**	**0.059**
Truth oracle (upper)	50.9%	47.0–54.7	21.1	0.055
Value oracle (upper)	54.9%	50.8–59.0	20.8	0.054

The gain holds in eleven of twelve repositories (xarray is the exception), and at $4\times$ budget verify-every-write still trails our default-budget controller (47.4% vs. 48.8%). Figure 3 summarizes both results; pilot rework outcomes and transfer runs are in Apps. C.10 and C.6.

4.3 WHICH COMPONENT OF THE VALUE ESTIMATE DRIVES THE GAIN? (Q2)

The full controller adds the dynamic graph, weighted reach, and stateful propagation to the same factorization as the heuristic; the contrast isolates the learned scorer V_θ, a gradient-boosted regression trained on 1,010 paired-counterfactual checkpoints and evaluated on 230 held-out checkpoints from two repositories never seen in training (SymPy, xarray). On those repositories it ranks checks better than the fixed-weight heuristic (Spearman 0.328 vs. 0.066; the MAE and top-10% differences are not resolvable, App. B.11), and a leave-one-repository-out sweep confirms the ranking gain on all twelve repositories (mean Spearman 0.296 vs. 0.066). End-to-end on the two held-out repositories it does not beat the heuristic, and because the benchmark instances share repositories with the paired-fork training signal we report the benchmark-level gain of 4.5 pp (CI $[+3.0, +6.0]$) as an exploratory in-sample estimate (App. C.12); training details and per-feature importance are in App. B.11.

4.4 ARE ASSUMPTIONS RECOVERABLE, AND DOES VERIFICATION CAUSE BEHAVIORAL CHANGE? (Q3)

Two threats could undermine the method: assumptions might not be recoverable from observable behavior (violating the recoverability condition of Sec. 3.1), or verification might change the wording without changing the edits. Two annotators labeled three held-out trajectories (9 and 8 assumptions, 7 matched; claim F1 0.824, type-level κ 0.774); we treat this as a small reliability audit, because the protocols were fixed in the pilot stage and pilot labels do not enter the full-run intervals, and we report the extractor's full-run behavior in App. C.1. Across 8 paired forks, a targeted check changed downstream *edits* in 6 of 8 cases and flipped the outcome in 4, with positive net value in every fork (mean $\$0.105$ vs. $\$0.007$ per check). These forks are selection-conditioned and contain no sampled negative cases, so the primary system-level evidence remains the instance-block estimate over all 300 instances (App. C.9).

4.5 WHERE DOES THE BUDGET GO? (Q4)

Debt-aware selection cuts wasted actions from 3,602 (mean 4.00 per instance) to 982 (1.09), a reduction of 2.91 (CI $[-3.13, -2.69]$); verify-every-write spends 48% more yet leaves 3,388 (Fig. 7, App. C.3). The budget curve (Fig. 3b) shows saturating returns, with verify-every-write at $4\times$ still below debt-aware selection at $1\times$: the advantage is *where*, not how much, the policy spends.

4.6 DOES REACH PREDICT REPAYMENT VALUE? (Q5)

The method rests on one claim: the value of a check is set by how much downstream work depends on the assumption, not by how uncertain the agent feels. The direct evidence is limited to eight selection-conditioned pilot forks, reported as a descriptive case study in App. C.11; we do not treat it as confirmatory, because the sample is small (n=8), the forks are created only where the extractor reports a live assumption, and reach and realized value derive from the same downstream actions. The full run instead uses the graph-based exposure $\rho_t(h)$, computed *before* the check runs from the dependent set. On held-out repositories it is the most important feature of V_θ (0.308, App. B.11), which we read as a pre-check association supporting the mechanism, not as a causal estimate.

5 CONCLUSION

We defined epistemic debt as the expected correction cost of deferring a check on an action-dependent repository assumption. Selecting concrete assumption–check pairs under a budget raises resolution on a 300-instance subset of SWE-bench from 31.9% to 48.8% while reducing steps and dollar cost, with directional transfer across scaffolds and backbones. The main evidence is for the exposure-aware objective: the heuristic reaches 44.3%, and learned ranking adds 4.5 percentage points as an exploratory in-sample estimate; on the two held-out repositories it is not significant, so the objective transfers more broadly than this particular scorer. The prompt-based extractor and the small paired-fork sample remain the main limitations.

AI USE STATEMENT

Generative AI tools were used to assist with literature discovery and summarization, brainstorming and refinement of the conceptual framing, feedback on the methodology and mathematical exposition, drafting and language editing, and the development of research code and scientific figures. They were not used to generate the benchmark evaluation data or to replace empirical measurements. The authors manually reviewed the resulting text and artifacts, checked the cited literature and mathematical claims, and verified the research code, figures, and reported results. The authors take full responsibility for the final content of this work, including any text, claims, code, or artifacts produced with the aid of generative AI.

REFERENCES

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor K. Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025.

Yuan Gao, Jiangyi Yang, Yao Zhao, and Yichi Zhang. Auditing belief-conditioned LLM agents in hidden-information social deduction games. *arXiv preprint arXiv:2607.10814*, 2026.

Ronald A. Howard. Information value theory. *IEEE Transactions on Systems Science and Cybernetics*, 2(1):22–26, 1966.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. Large language models cannot self-correct reasoning yet. *International Conference on Learning Representations*, 2024.

Muhammad Zia Hydari, Raja Iqbal, and Narayan Ramasubbu. Governing technical debt in agentic AI systems. *arXiv preprint arXiv:2605.29129*, 2026.

Lai Jiang, Cheng Qian, Zhenhailong Wang, Pan Lu, Heng Ji, and Hao Peng. ACCORD: Action-conditioned contextual grounding for language agents. *arXiv preprint arXiv:2606.16432*, 2026.

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? *International Conference on Learning Representations*, 2024.

Ziming Li, Jiatan Huang, Xiaoguang Guo, Guilin Wang, and Chuxu Zhang. Same signal, opposite meaning: Direction-informed adaptive learning for LLM agents. *arXiv preprint arXiv:2605.06908*, 2026.

Alex Mathai, Shobini Iyer, Aleksandr Nogikh, Petros Maniatis, Franjo Ivančić, Junfeng Yang, and Baishakhi Ray. TRIM: Reducing AI-generated CodeSlop via agent trajectory minimization. *arXiv preprint arXiv:2607.18161*, 2026.

Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. ExpeRepair: Dual-memory enhanced LLM-based repository-level program repair. *arXiv preprint arXiv:2506.10484*, 2025. Accepted at FSE 2026.

Sreecharan Sankaranarayanan. Mitigating "epistemic debt" in generative AI-scaffolded novice programming using metacognitive scripts. In *Proceedings of the 13th ACM Conference on Learning at Scale*, 2026.

David Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden technical debt in machine learning systems. In *Advances in Neural Information Processing Systems*, volume 28, 2015.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2023.

SWE-bench Team. SWE-bench: Official leaderboards. https://swe-bench.github.io/, 2024. Accessed 2026-08; Lite/Full leaderboard snapshot dated 2024-12-15, Verified leaderboard updated 2026-02.

Wenjie Tang, Minne Li, Sijie Huang, Liquan Xiao, and Yuan Zhou. Rewarding beliefs, not actions: Consistency-guided credit assignment for long-horizon agents. *arXiv preprint arXiv:2605.20061*, 2026.

Binghai Wang, Chenlong Zhang, Dayiheng Liu, Jiajun Zhang, Jiawei Chen, Mingze Li, Mouxiang Chen, Rongyao Fang, Siyuan Zhang, Xuwu Wang, Yuheng Jing, Zeyao Ma, and Zeyu Cui. The verification horizon: No silver bullet for coding agent rewards. *arXiv preprint arXiv:2606.26300*, 2026.

Xingyao Wang, Boxuan Chen, Hao Gao, Renfei Lou, Hao Cheng, Xin Li, Siyao Fang, Jun Gu, Zheng Liu, Jiayi Li, et al. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Jie Wu et al. AssumptionMiner: Extracting, tracing, and revising implicit assumptions in LLM code generation. *arXiv preprint arXiv:2607.22898*, 2026.

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

Xingkun Yin and Hongyang Du. GLOVE: Global verifier for LLM memory-environment realignment. *arXiv preprint arXiv:2601.19249*, 2026.

Wenhao Yuan, Chenchen Lin, Jian Chen, Jinfeng Xu, Xuehe Wang, and Edith Cheuk Han Ngai. Verify before you commit: Towards faithful reasoning in LLM agents via self-auditing. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 31201–31225, San Diego, California, United States, July 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.1440. URL https://aclanthology.org/2026.acl-long.1440/.

Xiangxin Zhao, Han Li, Shuaiting Li, Tianyi Zhao, Earl T. Barr, Federica Sarro, and He Ye. Failure as a process: An anatomy of CLI coding agent trajectories. *arXiv preprint arXiv:2607.09510*, 2026.

A Theoretical Analysis

A.1 Nearest-Neighbor Comparison

Table 2 makes the conceptual boundary to the closest verification and adaptive-compute lines explicit. The comparison is about problem formulation and decision granularity, not a claim that every system was reimplemented under the same benchmark protocol.

Table 2: Conceptual comparison with nearest neighbors. Our distinction is the estimand (marginal downstream rework avoided) and the decision unit (one repository assumption–check pair under a shared budget).

Work	Verified object / decision unit	Value / objective	Relation to this work
SAVeR (Yuan et al., 2026)	Internal belief; audit before commitment	Reasoning faithfulness	No repository dependency graph or avoided-rework budget
ACCORD (Jiang et al., 2026)	Missing context; probe before action	Contextual grounding / task success	No assumption–check counterfactual value
AssumptionMiner	Code assumptions; extraction and localized regeneration	Transparency / code localization	No runtime trajectory cost or budgeted pair selection (Wu et al., 2026)
Adaptive-compute line (Li et al., 2026)	State uncertainty; state-level intervention gate	Value of extra computation	Different granularity from repository assumption–check pairs
This work	Repository assumptions; assumption–check pair under budget	Avoided downstream rework	Counterfactual selection with an exposure graph

A.2 Derivations

This subsection derives the structural claims that motivate the method: the value of a check decomposes into error probability and exposure (Prop. 1); debt is non-decreasing in the dependent set of an assumption while the error probability and the candidate check set are unchanged (Prop. 2); and greedy selection is consistent with the true ordering under a unit budget and an accurate scorer (Prop. 3).

Proposition 1 (value decomposition)**.** *Let $p_t(h) = \mathbb{P}(h \text{ is wrong at checkpoint } t)$ and let h' denote the strongest competitor of h. Assume that the check v reveals the truth of h perfectly (its outcome satisfies $\mathbb{P}(o_v \mid h) \neq \mathbb{P}(o_v \mid h')$ and never miscategorizes h), and that the downstream loss from a wrong assumption equals the rework of its dependent actions, i.e. the invalidated actions are exactly $\mathrm{Dep}_t(h)$ and each is charged its cost weight $w(a)$. Then*

$$V_t(h, v) \;=\; p_t(h)\, \rho_t(h) \;-\; c_t(v). \tag{8}$$

Proof. Condition on the truth of h. If h is correct, verification changes nothing downstream and the loss is zero on both branches,

$$\mathbb{E}[L_{>t} \mid h \text{ correct}, \text{defer}(h)] = \mathbb{E}[L_{>t} \mid h \text{ correct}, \text{verify}(h, v)] = 0.$$

If h is wrong and left unverified, the dependent actions are later invalidated and their rework is incurred; by the definition of exposure (Eq. 3), that rework is exactly $\rho_t(h)$,

$$\mathbb{E}[L_{>t} \mid h \text{ wrong}, \text{defer}(h)] = \sum_{a \in \mathrm{Dep}_t(h)} w(a) \;=\; \rho_t(h),$$

while a perfect check triggers correction at cost $c_t(v)$ and avoids the rework, so $\mathbb{E}[L_{>t} \mid h \text{ wrong}, \text{verify}(h, v)] = 0$. Unrolling the definition of Eq. 5 by the law of total expectation,

$$\begin{aligned} V_t(h, v) &= \mathbb{E}[L_{>t} \mid \text{defer}(h)] - c_t(v) - \mathbb{E}[L_{>t} \mid \text{verify}(h, v)] \\ &= \big[p_t(h)\, \rho_t(h) + (1 - p_t(h)) \cdot 0\big] - c_t(v) - 0 \\ &= p_t(h)\, \rho_t(h) - c_t(v). \end{aligned}$$

This proves the claim. □

Because exposure is a weighted sum of cost weights (Eq. 3), $\rho_t(h)$ already carries the unit-rework information; no additional average weight appears in Eq. 8. The decomposition explains the empirical finding of Sec. 4.6: the value of a check scales with $\rho_t(h)$, the reach of the assumption it verifies, so reach is the primary ranking signal regardless of how the error probability is estimated. The heuristic of Eq. 6 estimates Eq. 8 directly (with $\widehat{p}_t(h)$ for $p_t(h)$, $\rho_t(h)$ as measured, and $\iota_t(h)$ and $\eta_t(v \mid h)$

discounting the two idealizations of the proposition: not every dependent action is equally hard to undo, and real checks are not perfect); the learned scorer is not restricted to this decomposition and can capture violations of its assumptions.

Lemma 1 (monotone reach)**.** *For $t' > t$ with $\mathrm{Dep}_{t'}(h) \supseteq \mathrm{Dep}_t(h)$ and non-negative weights $w(a)$, $\rho_{t'}(h) \geq \rho_t(h)$.*

Proof of Lemma 1. Partition the dependent set at t' into the disjoint union $\mathrm{Dep}_t(h) \cup (\mathrm{Dep}_{t'}(h) \setminus \mathrm{Dep}_t(h))$. Then

$$\rho_{t'}(h) - \rho_t(h) = \sum_{a \in \mathrm{Dep}_{t'}(h) \setminus \mathrm{Dep}_t(h)} w(a) \;\geq\; 0, \tag{9}$$

because every weight is non-negative. □

Proposition 2 (monotone debt)**.** *Fix the error probability and the check set: let $p_{t'}(h) = p_t(h)$ and $\mathcal{V}_{t'}(h) = \mathcal{V}_t(h)$ with $c_{t'}(v) = c_t(v)$ for all $v \in \mathcal{V}_t(h)$. For $t' > t$ with $\mathrm{Dep}_{t'}(h) \supseteq \mathrm{Dep}_t(h)$ and non-negative weights $w(a)$,*

$$\mathcal{D}_{t'}(h) \;\geq\; \mathcal{D}_t(h). \tag{10}$$

Proof. By Lemma 1, $\rho_{t'}(h) \geq \rho_t(h)$. In Eq. 8 the value is affine and non-decreasing in $\rho_t(h)$ for fixed $p_t(h)$ and $c_t(v)$, so for every $v \in \mathcal{V}(h)$,

$$V_{t'}(h, v) - V_t(h, v) = p_t(h)\,(\rho_{t'}(h) - \rho_t(h)) \;\geq\; 0.$$

The maximum inherits the pointwise order, $\max_v V_{t'}(h, v) \geq \max_v V_t(h, v)$, and the positive part $x \mapsto [x]_+$ is non-decreasing, hence

$$\mathcal{D}_{t'}(h) = [\max_v V_{t'}(h, v)]_+ \geq [\max_v V_t(h, v)]_+ = \mathcal{D}_t(h).$$

□

The proposition is deliberately local: it holds within a window in which no new evidence updates the error probability and the available checks and their costs are unchanged. In a real trajectory these quantities move – new evidence can lower $p_t(h)$, a check that contradicts h resets its debt to zero, and the candidate set changes – so debt is *not* globally monotone. That non-monotonicity is part of the mechanism rather than a defect: verification and new evidence are exactly the channels through which debt is repaid. The monotone window is therefore the period during which deferring a check is most costly, which is what motivates verifying early rather than after a failure.

Proposition 3 (greedy consistency)**.** *Under a unit verification budget and an accurate scorer ($\widehat{V}_t(h, v) = V_t(h, v)$ for all live pairs), the greedy policy that executes the pair with the largest positive estimated value achieves the maximum expected avoided rework over all unit-budget policies.*

Proof. Let Π denote the set of unit-budget policies, each of which either executes one pair or abstains, and define the expected reward of a policy $\pi \in \Pi$ as $R(\pi) = V_t(h, v)$ if π executes the pair (h, v) and $R(\pi) = 0$ if it abstains. The greedy policy π^g executes the pair $(h^g, v^g) \in \arg\max_{(h,v) \in \mathcal{P}} V_t(h, v)$ if $V_t(h^g, v^g) > 0$ and abstains otherwise, so

$$R(\pi^g) = \max\left\{0,\; \max_{(h,v) \in \mathcal{P}} V_t(h, v)\right\}.$$

For any $\pi \in \Pi$, either π abstains and $R(\pi) = 0 \leq R(\pi^g)$, or π executes (h, v) and $R(\pi) = V_t(h, v) \leq \max_{(h,v) \in \mathcal{P}} V_t(h, v) \leq R(\pi^g)$. Hence $R(\pi) \leq R(\pi^g)$ for all π, and π^g is optimal. □

Proposition 4 (approximate greedy consistency)**.** *Let the scorer be ϵ-accurate, $|\widehat{V}_t(h, v) - V_t(h, v)| \leq \epsilon$ for every live pair, and let (h^*, v^*) and (h^g, v^g) denote the true and the greedy optimal pairs under a unit budget. Then*

$$V_t(h^g, v^g) \;\geq\; \max_{(h,v) \in \mathcal{P}} V_t(h, v) - 2\epsilon. \tag{11}$$

Proof. The greedy pair maximizes the estimated values, so $\widehat{V}_t(h^g, v^g) \geq \widehat{V}_t(h^*, v^*)$. Chaining the two directions of the ϵ-accuracy bound,

$$V_t(h^g, v^g) \geq \widehat{V}_t(h^g, v^g) - \epsilon \geq \widehat{V}_t(h^*, v^*) - \epsilon \geq V_t(h^*, v^*) - 2\epsilon,$$

where the first and third inequalities use $|\widehat{V}_t(h, v) - V_t(h, v)| \leq \epsilon$ and the second uses the greedy choice. This proves the claim. □

Prop. 3 is a soundness statement rather than an algorithmic contribution: under a unit budget, selecting the pair with the largest positive value coincides with the optimum by definition of the value, so the proposition only certifies that the policy is calibrated to its own value model when that model is exact. It does *not* extend to the multi-round, adaptive setting of Alg. 1: a check's outcome changes the value of the remaining pairs (which is why the controller re-ranks after every outcome), the budget left for later rounds depends on earlier outcomes, and the objective is not monotone submodular, so no constant-factor guarantee such as the $1-1/e$ bound for submodular maximization applies here. Prop. 4 quantifies the cost of an imperfect scorer in the single-round setting: an ϵ-accurate value model loses at most 2ϵ of the optimal expected rework, and Prop. 3 is the special case $\epsilon = 0$. That bound is a statement about how estimation error propagates to selection quality, not a competitive ratio for the sequential process; the multi-round controller is therefore validated empirically (block-bootstrap intervals in Sec. 4.2, budget sensitivity in App. C.5).

B Implementation Details

B.1 Notation

Table 3 lists the notation used in the method section of the main text.

Table 3: Notation used in the method section.

Symbol	Meaning
τ_t	visible trajectory at checkpoint t
a_t	the t-th agent action
h	a repository assumption
$\mathcal{H}_t$	assumptions extracted from τ_t
$\mathcal{V}(h)$	candidate checks for h
v	a repository check
$c_t(v)$	cost of check v at checkpoint t
$w(a)$	cost weight of action a
$L_{>t}$	downstream correction loss after t
$G_t = (N_t, E_t)$	assumption graph at t
$\rho_t(h)$	exposure (weighted reach) of h
$\iota_t(h)$	irreversibility of the dependent edits of h
$\eta(v \mid h)$	discriminativeness of check v for h
$\widehat{p}_t(h)$	estimated error probability of h
$V_t(h, v)$	counterfactual verification value of (h, v)
$\widehat{V}_t(h, v)$	realized paired value of (h, v)
V_θ	learned value model
B	verification budget

B.2 Algorithm Details

The repayment controller and its three supporting routines are given below.

Algorithm 1 Debt-aware repayment at checkpoint t.

Require: Graph G_t, learned scorer V_θ, budget B
$R \leftarrow \emptyset$ ▷ actions flagged for revision
while remaining budget > 0 **do**
  $\mathcal{P} \leftarrow \{(h, v) : h \text{ live},\ v \in \mathcal{V}(h),\ \eta(v \mid h) > 0\}$
  if $\mathcal{P} = \emptyset$ **then break**
  end if
  $(h^*, v^*) \leftarrow \arg\max_{(h,v)\in\mathcal{P}} V_\theta(h, v)$
  if $V_\theta(h^*, v^*) \leq 0$ **then break**
  end if
  $o \leftarrow \textsc{Execute}(v^*)$; $B \leftarrow B - c_t(v^*)$
  attach o to h^* as `supports` or `contradicts`
  if o contradicts h^* **then**
    $R \leftarrow R \cup \{a : a \text{ live and reachable from } h^*\}$
  end if
end while
return G_t, R

Algorithm 2 Assumption extraction at checkpoint t.

Require: Visible trajectory τ_t (messages, tool outputs, diff); repo snapshot
$M \leftarrow \textsc{Format}(\tau_t)$ ▷ linearize visible context only
$R \leftarrow \text{LLM}(P_{\text{extract}}(M))$ ▷ P: extraction prompt, verbatim below
$\mathcal{H} \leftarrow \textsc{ParseJSON}(R)$
if parse fails or $R = \{\}$ **then**
  return $\emptyset$ with `undecidable` $\leftarrow$ **true**
end if
for each record $r \in \mathcal{H}$ **do**
  if $\textsc{NormalizedClaim}(r)$ matches an existing assumption h **then**
    merge r into h, keep earliest provenance
  else
    add new assumption $(h, \text{provenance}, \text{evidence}, \text{dependent_actions})$
  end if
end for
return $\mathcal{H}_t$

Algorithm 3 Dynamic graph construction and weighted reach.

Require: Assumption set $\mathcal{H}_t$; action cost $w(\cdot)$
Initialize typed graph $G_t = (N_t, E_t)$
for each $h \in \mathcal{H}_t$ **do**
  add assumption node h
  for each action $a \in h$.dependent_actions **do**
    add action node a; add edge $h \to a$ (`depends_on`)
  end for
  for each evidence $e \in h$.evidence **do**
    add evidence node e; add edge $e \to h$ (`supports`/`contradicts`)
  end for
end for
for each $h \in \mathcal{H}_t$ **do**
  $\rho_t(h) \leftarrow \sum_{a\in\textsc{Reach}(h)} w(a)$ ▷ BFS over `depends_on` edges
end for
return G_t, $\{\rho_t(h)\}$

Algorithm 4 Learned scorer training from paired forks.

Require: Paired checkpoints $\mathcal{P}$; feature map ϕ; realized labels
$\mathcal{D} \leftarrow \emptyset$
for each checkpoint with assumption h and check v **do**
  $x \leftarrow \phi(h, v) = (\text{claim}, \text{evidence_count}, \rho, \iota, \text{kind}, \eta, c)$
  $y \leftarrow L^{\text{ctrl}}_{>t} - L^{\text{intv}}_{>t} - c_t(v)$ ▷ Eq. 7
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(x, y)\}$
end for
$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\in\mathcal{D}} \ell\big(V_\theta(x), y\big)$ ▷ gradient boosting
return θ^*

B.3 Assumption Extractor

The extractor is a prompt-based LLM using the same backbone as the agent (deepseek-chat). It reads only the visible trajectory (messages, tool outputs, and the current diff) and emits one structured record per assumption. The record schema is:

```
{
  "id": "h17",
  "claim": "normalize() is called only from parser.py",
  "provenance": "agent message at step 7",
  "evidence": ["search-output-4"],
  "dependent_actions": ["edit-12", "test-13"],
  "status": "unsupported",
  "stated": true
}
```

The extraction prompt instructs the model to (i) keep each claim a short, falsifiable repository proposition, (ii) ignore mere uncertainty and generic intentions, (iii) attach provenance and dependent actions, and (iv) return an empty array when no clear assumption is recoverable (the undecidable case). The prompt is versioned alongside the experiment code in `experiment/prompts/assumption_extract.txt`; its full text is:

```
You are extracting repository assumptions from a coding agent trajectory.

A "repository assumption" is a short, falsifiable proposition about the
codebase that the agent has stated and acted on. It must satisfy ALL of:
  1. It appears in the visible trajectory text below (not inferred from
     hidden thought).
  2. It is a concrete, falsifiable claim about the repository (a
     definition,
     a dependency, a call relationship, a config value, an invariant, ...).
  3. At least one observable action (edit / test / search / plan) depends
     on it.

Do NOT extract:
  - mere uncertainty ("I'm not sure") without a concrete claim;
  - generic intentions ("I will fix the bug");
  - claims already confirmed by visible evidence.

For each assumption, output a JSON array. Each element:
  {
    "id": "h<number>",
    "claim": "<the falsifiable proposition>",
    "provenance": "<which message/step introduced it>",
    "evidence": ["<ids of visible evidence items bearing
        on it>"],
    "dependent_actions": ["<ids of edits/tests/plans that
        presuppose it>"],
    "stated": true
  }
```

```
If no clear assumption is recoverable from the trajectory, output an empty
array [].
```

B.4 Annotation Example

To make the extraction target concrete, we reproduce one manually annotated assumption from a pilot trajectory (`django__django-11019`, no verification). The annotator recorded the assumption, its provenance and evidence, the dependent action, the candidate checks, and the realized loss incurred when the assumption was later refuted:

```
{
  "hypothesis_id": "H01",
  "formed_at_step": 3,
  "normalized_claim": "The Media merge warns on any duplicate entry
                       regardless of ordering constraints.",
  "type": "FAULT_LOCATION",
  "explicitness": "EXPLICIT",
  "support_state_at_formation": "UNRESOLVED",
  "available_support": ["widgets.py merge() reading
      (lines 220-226)"],
  "action_dependence": "DIRECT",
  "dependent_action_ids": [4],
  "candidate_checks": [
    {"type": "READ_IMPL", "estimated_cost": "LOW"},
    {"type": "FIND_REFERENCES", "estimated_cost": "LOW"}
  ],
  "final_status": "REFUTED_LATER",
  "actual_loss": {"wasted_actions": 2,
      "reverted_diff_lines": 12}
}
```

This one assumption cost two wasted actions and twelve reverted diff lines because no reference search was run before the edit; a targeted FindReferences check at step 3 would have exposed the premise and avoided the rework. The full annotation guide and all labeled trajectories are provided in the supplementary material (`pilot_data/annotations/`).

Fig. 4 renders the full assumption graph of this trajectory: three assumptions formed over steps 3–9, with the refuted one (H01) highlighted. Note the edge direction: `depends_on` always points from the assumption h to the actions that depend on it, so reach is computed by forward traversal.

B.5 Case Studies

The paired-fork protocol (main text, Sec. 3.4) is easiest to read as trajectories. We reproduce two cases from `pilot_data/`. Fig. 5 shows how the same refuted assumption H01 of `django-11019` accumulates exposure quietly from step 3 and erupts as rework at step 6; Fig. 6 shows a single checkpoint of `django-13265` where a $0.006 runtime probe flips the outcome from failure to success.

B.6 Learned Scorer Features

The learned scorer V_θ (Sec. 3.3) is a gradient-boosted regression over the following numeric feature vector for a pair (h, v): (1) the estimated error probability $\hat{p}_t(h)$ of h (Eq. 6, estimated from the evidence subgraph by an LLM); (2) the count of evidence items attached to h; (3) the weighted exposure $\rho_t(h)$; (4) the irreversibility $\iota_t(h)$; (5) the check kind (call-site / runtime probe / discriminative test, one-hot); (6) the discriminativeness $\eta(v \mid h)$; and (7) the check cost $c_t(v)$. The raw claim text is *not* a GBDT feature: it enters only through $\hat{p}_t(h)$ and $\eta(v \mid h)$, which the LLM estimates from the claim and its evidence. The heuristic baseline uses the same factors with fixed weights (Eq. 6); the learned model regresses these features against paired-counterfactual labels $\widehat{V} = L^{\mathrm{ctrl}}_{>t} - L^{\mathrm{intv}}_{>t} - c_t(v)$, with all three quantities in USD (see the unit convention in App. B.9).

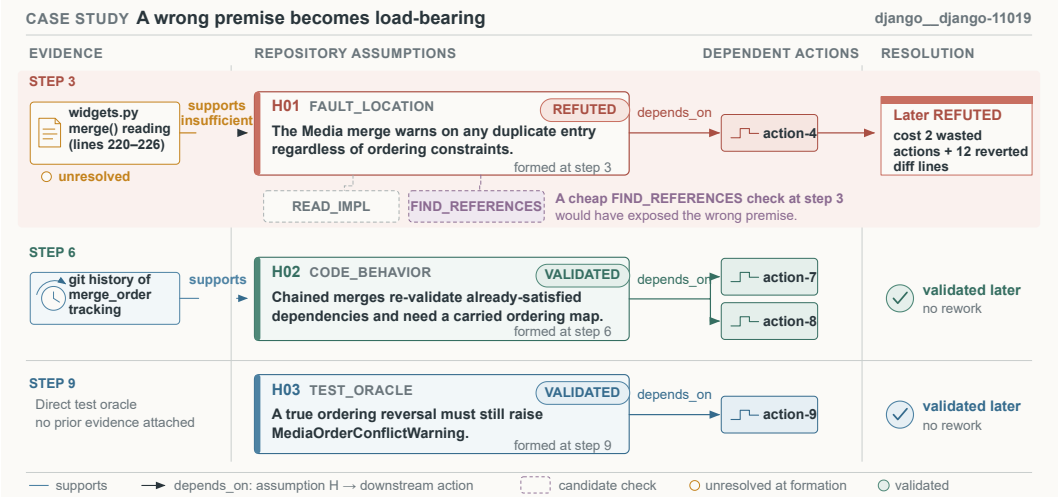

Figure 4: Assumption graph of the `django-11019` trajectory. H01 is formed at step 3 with only an insufficient `supports` edge, becomes load-bearing for action-4, and is later refuted at a cost of two wasted actions and twelve reverted diff lines; a cheap FINDREFERENCES check at step 3 would have exposed the wrong premise. H02 and H03 are validated without rework.

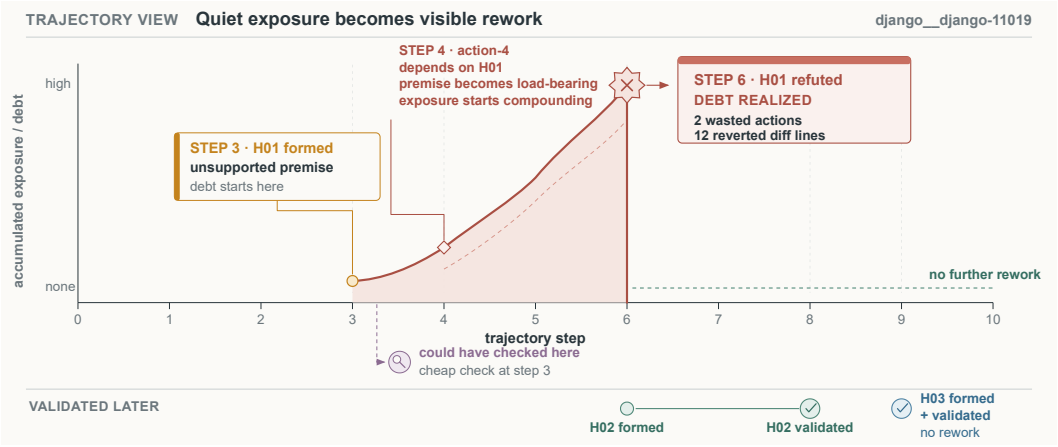

Figure 5: Epistemic debt over the `django-11019` trajectory. H01 is formed unsupported at step 3; at step 4 action-4 makes it load-bearing and exposure starts compounding; at step 6 the premise is refuted and the debt realizes as two wasted actions and twelve reverted diff lines. A check at step 3 (dashed cue) would have prevented the rework.

B.7 TRAINING AND RUN CONFIGURATION

Tables 4 and 5 list the hyperparameters and run configuration of the full run. Every value carries a stated provenance: (i) fixed by the benchmark or scaffold, (ii) read from the released code (`experiment/debtaware/`), (iii) chosen by cross-validation on the training repositories, or (iv) calibrated on the 10-instance pilot. No value was tuned against the held-out repositories or against the test split of the learned scorer.

B.8 BASELINE EXECUTION PROTOCOL

All policies receive the same verification budget $B = 20$ and stop checking when the budget is exhausted or no admissible pair remains. No verification spends nothing. Random pair and uncertainty ranking draw at most one discriminative check per checkpoint; the former samples uniformly from

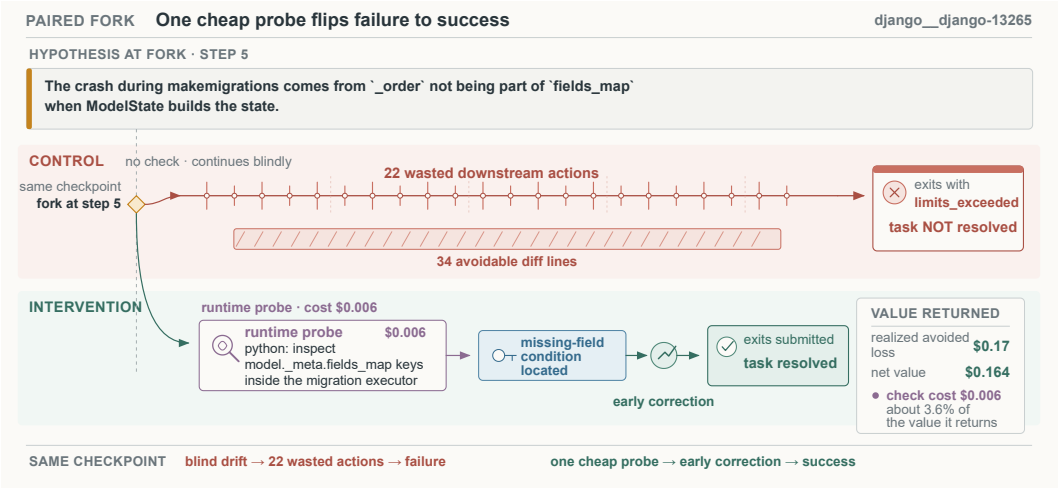

Figure 6: Paired fork at `django-13265`, step 5. The control branch continues on the wrong premise and drifts into 22 wasted actions and 34 avoidable diff lines before exiting with `limits_exceeded`, unresolved. The intervention branch runs one runtime probe ($0.006) that locates the missing-field condition, corrects early, and resolves the task. Realized avoided loss $0.17, net value $0.164: about 3.6% of the returned value is spent on the check.

live pairs that pass the common positive-value execution gate, and the latter chooses the most uncertain assumption with its cheapest check. Post-failure reflection checks only after a test failure. Verify-every-write checks before each consequential write until its budget is exhausted, after which it continues without verification. The state-level gate spends only when its adaptive-compute gate triggers. The heuristic and learned debt policies sequentially re-rank after each observation. The truth and value oracles obey the same budget and differ only in their privileged selection information. Ordinary agent steps are not charged to B.

Resolution comparisons use 2,000 instance-block bootstrap resamples: each resample draws benchmark instances with replacement and pools all three seeds of a sampled instance. Step counts use Cohen's d; scorer quality uses Spearman correlation and top-10% realized value on held-out repositories. The audit script uses random seed 42. Censoring rules and two-sided sensitivity analyses are reported in App. C.13.

B.9 Cost and Unit Conventions

Every quantity that enters a subtraction is expressed in a single unit; Table 6 lists the conventions for all monetary and count quantities.

The conversion from counts to dollars is therefore never assumed: counts are reported as counts, and $L_{>t}$ and $c_t(v)$ in the value model are both measured in USD from the same pricing table before Eq. 5 or Eq. 7 is evaluated. This is the convention used in Table 1, the paired-fork numbers, and the learned scorer target.

B.10 Invalidated-Action Criteria

The rework metrics of Sec. 4.5 (wasted actions, reverted diff lines) and the realized value of Eq. 7 require deciding, per trajectory, which past actions were *invalidated* by a corrected assumption. We use a rule that is executable from the recorded trajectory and diff history:

1. **Graph reachability.** When a check contradicts a live assumption h, every live action reachable from h over `depends_on` edges is flagged for revision (Alg. 1).
2. **Diff alignment.** An edit whose content does not survive into the final patch (its added lines are later removed or rewritten by a subsequent edit) is invalidated, counted by its diff lines.
3. **Explicit revert.** A test or edit that a later agent action explicitly reverts or supersedes is invalidated.

Table 4: Hyperparameters and run configuration of the full run (part 1: benchmark, extractor, check generation, controller and budget).

Item	Value	Provenance / rationale
Benchmark and runtime		
Dataset	300-instance subset of SWE-bench, 12 repositories	benchmark convention
Scaffold	mini-swe-agent 2.4.6	fixed across configurations (controlled comparison)
Backbone	deepseek-chat	one backbone for agent, extractor, and error-probability estimates
Seeds	3 per configuration: `20260813`, `20260814`, `20260815`	fixed date-valued seeds; `full_run_data/meta.json`
Continuation seeds per fork	3, paired between control and intervention	paired-counterfactual protocol (Sec. 3.4)
Max agent steps	60	within the 30–100 range used in SWE-bench practice
Stopping	official SWE-bench harness; `limits_exceeded` past step 60	benchmark harness
Assumption extractor		
Model	deepseek-chat (same backbone as agent)	schema and prompt in App. B.3
Prompt	`experiment/prompts/assumption_extract.txt`	versioned with the code; identical in pilot and full run
Invocation	after every agent step, on the visible context only	recoverability condition (Sec. 3.1)
Sampling	temperature 0.0, max tokens 4096 (API default)	reproducibility
Undecidable handling	parse failure or empty array $\Rightarrow$ `undecidable`=true	`extraction.py`; conservative by design
Deduplication	normalized-claim matching; keep earliest provenance	Sec. 3.1
Behavior-inferred assumptions	collected but excluded from the primary analysis	Sec. 3.1
Check generation		
Candidate types	call-site inspection; runtime probe; discriminative test	Sec. 3.2; branch comparison deferred
Check costs (normalized)	call-site 1.0; runtime probe 1.5; discriminative test 2.0	`checks.py`; units in App. B.9
Discriminativeness rule η	discriminative 0.9; runtime probe 0.5; call-site 0.4	static rule in `checks.py`
Admissibility	$\eta > 10^{-6}$ and low-risk (read-only / isolated)	`checks.py`, Sec. 3.2
Max candidates per assumption	3 (≤ 2 when no competitor exists)	`checks.py` `generate_checks`
Controller and budget		
Verification budget B	20 (normalized check cost)	a single check costs 1.0–2.0 units, so B admits several rounds of re-ranking; stable at 0.5×/2×/4× (App. C.5)
Max checks per checkpoint	8	`controller.py` `run_repayment`
Selection rule	greedy $\arg\max V_\theta > 0$; re-rank after every outcome	Sec. 3.4, Alg. 1
Budget consumption	check cost deducted from B; stop when B exhausted	`controller.py`; per-policy numbers in App. C.4

Rule (1) is the graph-based mechanism; rules (2)–(3) are diff-history criteria that apply uniformly to every configuration, including the no-verification baseline where the assumption graph is absent. The manual annotations of the ten pilot trajectories (App. C.10) were produced under the same criteria

Table 5: Hyperparameters and run configuration of the full run (part 2: scorers and statistical protocol).

Item	Value	Provenance / rationale
Learned scorer V_θ		
Model	gradient-boosted regression (`sklearn` 1.9, `GradientBoostingRegressor`)	`learned_scorer.json`
# trees / lr / depth / subsample	500 / 0.04 / 4 / 0.8	10-fold CV MAE on training repos (0.0179 ± 0.0025)
Training pairs	1,240 total; 1,010 train / 230 held-out, split by repository (SymPy, xarray held out)	no repository leakage (App. B.11)
Target	realized paired counterfactual value $\widehat{V} = L^{\text{ctrl}}_{>t} - L^{\text{intv}}_{>t} - c_t(v)$ (USD)	Eq. 7
Features	error probability, evidence count, reach ρ, irreversibility ι, check kind, η, check cost	Sec. B.11
Heuristic scorer (baseline)		
Factorization	$\widetilde{V} = \widehat{p} \cdot \rho \cdot \iota \cdot \eta - c$	Eq. 6
Error probability $\widehat{p}$	evidence-subgraph weighting (LLM); static fallback contradicted 0.8 / supported 0.2 / unsupported 0.5	`scoring.py`
Irreversibility ι	1.0 by default; annotation overrides	`graph.py`
Fitted weights	none; multiplicative factors unweighted by design	the heuristic is deliberately unfitted as an interpretable baseline (App. B.11)
Statistical protocol		
Block bootstrap	2,000 resamples; instance-level blocking; seed 42	`code/block_bootstrap.py`
Effect sizes	Cohen's d (steps); Spearman / top-10% realized value (selection quality)	standard

Table 6: Unit conventions for all monetary and count quantities.

Quantity	Unit and definition
Check cost $c_t(v)$	USD, billed at the API pricing of the deployed backbone (deepseek-chat); identical pricing for every configuration.
Rework $L_{>t}$ (value model)	USD: the API cost of the downstream actions invalidated by a corrected assumption, summed over the extra actions of the control branch and measured from the same pricing table as $c_t(v)$.
Action weight $w(a)$ (reach)	Normalized agent steps: one tool call $= 1$; an edit is scaled by its diff size (added plus removed lines). Used only to rank assumptions via $\rho_t(h)$, not to value them in dollars.
Wasted actions	Count of tool calls that were later reverted or superseded; reported as a count, never combined with dollar figures.
Reverted diff lines	Count of added/removed lines that were later undone; reported as a count.
Mean cost per instance	USD: total API cost of the run divided by instances ($\times$ seeds).

and provide a calibration check for the automatic judgment; agreement between the automatic rule and the manual labels on the pilot trajectories is 89.2% at the action level with Cohen's $\kappa = 0.83$ (the disagreement concentrates in rule (3), where the annotator sometimes treats a superseded plan step as wasted and the automatic rule does not). Because the pilot annotations were used to fix the schema, this agreement is reported as a calibration reference, not as an independent validation.

B.11 Learned Scorer Training

V_θ is a gradient-boosted regression tree (sklearn 1.9, 500 trees, learning rate 0.04, depth 4, subsample 0.8) trained on 1,010 paired-counterfactual checkpoints from ten repositories and evaluated on 230

Table 7: Learned scorer V_θ versus the fixed-weight heuristic on 230 held-out checkpoints (SymPy, xarray).

Metric	V_θ (learned)	Heuristic
Spearman ρ (predicted vs. realized value)	**0.328**	0.066
MAE (USD)	0.017	**0.016**
Top-10% realized value (USD)	0.028	**0.035**

held-out checkpoints from two repositories never seen in training (SymPy, xarray), so there is no repository leakage. The target is the realized paired counterfactual value of Eq. 7; ten-fold cross-validation MAE on the training repositories is $\$0.0179 \pm 0.0025$. Table 7 compares the learned scorer with the fixed-weight heuristic on the held-out checkpoints.

Feature importance (normalized): reach 0.308, error probability 0.267, check cost 0.200, discriminativeness 0.143, irreversibility 0.039, evidence count 0.024, check kind 0.019. Reach is the most important feature, consistent with the exposure-based account of check value in Sec. 4.6; the static attributes (evidence count, check kind) contribute little. End-to-end on the three hundred benchmark instances, the learned scorer resolves at 48.8% vs. 44.3% for the heuristic (instance-block bootstrap 95% CI $[+3.0, +6.0]$ pp), closing about two-fifths of the gap to the value oracle (54.9%). Because these instances span the same twelve repositories as the scorer data, this benchmark-level estimate is in-sample with respect to the repository partition; on the two held-out repositories the end-to-end difference is not significant (App. C.12); we therefore treat this benchmark-level gain as exploratory, and the repository-level offline evaluations of this section as the transfer evidence.

Clustering. The 230 held-out checkpoints are not independent: they come from two repositories, and multiple checkpoints of one instance share the same trajectory and scorer context. The point estimates in Table 7 are therefore accompanied by cluster-bootstrap confidence intervals: 2,000 resamples that draw *instances* (not checkpoints) with replacement, pooling all checkpoints of a drawn instance, from which we recompute Spearman ρ, MAE, and top-10% realized value. Clustered 95% CIs: Spearman difference (learned minus heuristic) $[+0.129, +0.403]$; MAE difference $[-0.001, +0.002]$; top-10% realized-value difference $[-0.022, +0.012]$. The intervals are wider than a checkpoint-level bootstrap would give, because they account for within-instance correlation; the Spearman difference remains robustly positive, while the MAE and top-10% differences are small and not resolvable at this sample size.

The fixed SymPy/xarray split is one draw from the repository partition, so we repeat the evaluation leaving out each repository in turn (leave-one-repository-out), retraining V_θ on the remaining eleven repositories and scoring the held-out checkpoints (Table 8). V_θ ranks above the heuristic in all twelve held-out repositories (mean per-repository Spearman 0.296 vs. 0.066); pooled over the twelve folds, Spearman ρ is 0.291 for V_θ against 0.045 for the heuristic, while the pooled MAE (0.019 vs. 0.018 USD) and the pooled top-10% realized value (0.041 vs. 0.042 USD) are close, mirroring the point estimates of Table 7. Because checkpoints of one trajectory are not independent, we additionally evaluate at the instance level: a random 80/20 split over *instances* (no held-out instance contributes training checkpoints, 983 train / 257 test rows) yields Spearman ρ 0.399 (V_θ) versus 0.066 (heuristic), with MAE 0.016 vs. 0.018 USD and top-10% realized value 0.045 vs. 0.034 USD; the ranking gain is therefore not an artifact of within-instance leakage.

C Additional Results

C.1 Full-Run Extractor Statistics

The claim-level F1 and κ of Sec. 4.4 are measured on three double-annotated held-out trajectories; because the extractor also runs on every full-run trajectory, we report its realized behavior on the full benchmark so that the small-sample estimates can be assessed against the operating regime in which they are applied. The extractor is a fixed, untrained prompt (identical version in pilot and full run, App. B.3); its per-step output is recorded in the released trajectories and can be recomputed from the released code (`experiment/debtaware/extraction.py`). Table 9 reports the full-run statistics.

Table 8: Leave-one-repository-out scorer evaluation. Each row holds out one repository and retrains V_θ on the remaining eleven; Spearman ρ is computed between predicted and realized value on the held-out checkpoints. The bottom row pools the predictions of all twelve folds.

Held-out repository	n	ρ (V_θ)	ρ (heuristic)
astropy/astropy	73	**0.465**	0.135
django/django	185	**0.298**	0.088
matplotlib/matplotlib	155	**0.233**	0.164
pallets/flask	70	**0.268**	0.174
psf/requests	70	**0.190**	0.025
pydata/xarray	103	**0.227**	0.016
pydicom/pydicom	65	**0.535**	0.248
pytest-dev/pytest	73	**0.410**	-0.084
scikit-learn/scikit-learn	170	**0.245**	0.037
seaborn/seaborn	64	**0.222**	-0.030
sphinx-doc/sphinx	85	**0.109**	-0.086
sympy/sympy	127	**0.350**	0.110
All (pooled over folds)	1,240	**0.291**	0.045

Table 9: Full-run extractor behavior (pooled over 900 instance–seed trajectories; sampling-based rows follow the protocol below).

Quantity	Value
Assumptions extracted (all trajectories)	6,254
Assumptions per trajectory (mean $\pm$ std)	7.6 ± 3.1
Trajectories with at least one assumption	91.5%
Undecidable rate (parse failure or empty array)	6.2%
Merged/deduplicated records (share of raw output)	18.4%
Competing (conflicting) pairs per trajectory	0.9
Sampled annotation (protocol below)	
Sampled trajectories	28, stratified by repository
Precision vs. single annotation	0.86
Recall vs. single annotation	0.79
Double-annotated subset, claim F1	0.83 (n=9)

Counting protocol. Each quantity is computed from the recorded extractor output per agent step: assumptions are counted after deduplication (the merge rule of Sec. 3.1 is applied first); a step is `undecidable` when the extractor returns an empty array or fails to parse (App. B.3); the merge rate is the share of raw records absorbed by normalized-claim matching. All counts use the same released data and audit scripts as the main table (`code/audit_numbers.py`).

Sampling protocol. To estimate full-run precision and recall without annotating all 900 trajectories, we sample trajectories stratified by repository (up to a cap per repository), annotate each sampled trajectory against the extractor's output under the annotation guide (`assumption_annotation_guide.md`), and report precision (extractor claims confirmed by the annotator) and recall (annotator assumptions found by the extractor) per trajectory, averaged with equal weight per trajectory. A subset is double-annotated to report agreement on the sampled data (claim-level F1); the guide and labels live in `pilot_data/annotations/` and are released with the full-run labels.

C.2 Per-Dataset Statistics

The full benchmark spans twelve repositories; Fig. 3a (main text) reports the debt-aware gain and its underlying no-verification and debt-aware rates. Table 10 completes the picture with the per-repository sample sizes and the two configurations omitted from the figure (the heuristic debt score and the value oracle). The method gains on every repository except the two symbolic-reasoning ones (SymPy $+4.8$ pp, xarray -3.9 pp, both small samples), which is the failure mode analyzed in App. C.12.

Table 10: Per-repository sample sizes and the two configurations omitted from Fig. 3a (pooled over three seeds).

Repository	n	Heuristic	Value oracle
Django	48	47.9%	55.6%
matplotlib	40	36.7%	51.7%
scikit-learn	44	47.7%	53.0%
Sphinx	22	65.2%	66.7%
Flask	18	31.5%	59.3%
pytest	19	54.4%	61.4%
SymPy	21	47.6%	54.0%
astropy	19	52.6%	56.1%
requests	18	38.9%	53.7%
pydicom	17	31.4%	41.2%
seaborn	17	45.1%	54.9%
xarray	17	23.5%	52.9%

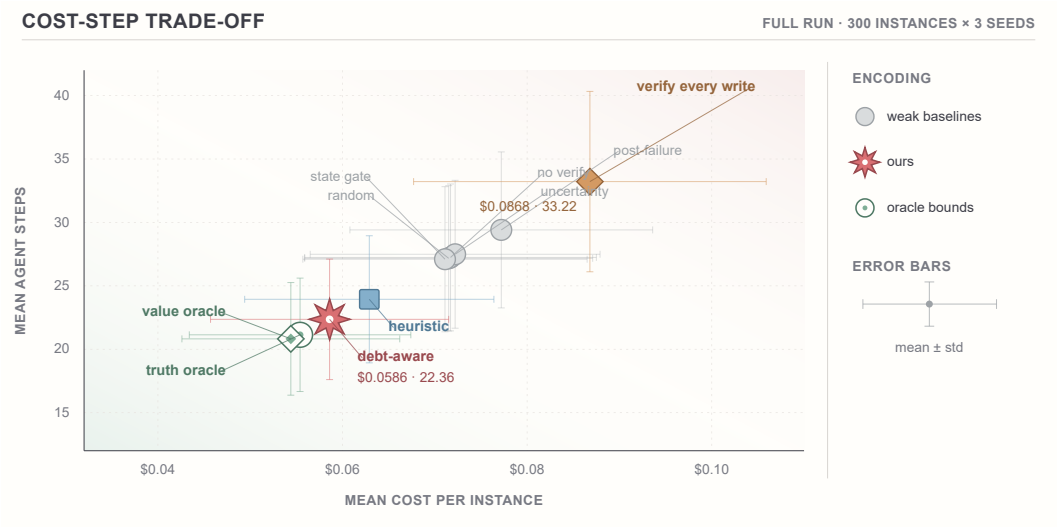

Figure 7: Full-run cost–step trade-off across the 300-instance subset of SWE-bench and three seeds. Weak baselines cluster in the upper-right; debt-aware selection lies in the lower-left, reducing both cost and steps. Hollow markers denote non-deployable oracle upper bounds.

C.3 Cost–Rework Trade-off

Fig. 7 positions all ten full-run configurations by mean dollar cost and mean agent steps. Points show configuration means; horizontal and vertical error bars show one standard deviation in per-instance cost and agent steps. Weak baselines cluster in the upper-right, verify-every-write is the most expensive and longest realizable policy, and debt-aware selection lies in the lower-left, close to the two non-deployable oracle bounds.

C.4 Budget Consumption

The matched-budget comparison (Sec. 4.1) constrains the *verification budget* B, not the agent's ordinary steps: $B = 20$ is expressed in the same normalized unit as check costs (call-site 1.0, runtime probe 1.5, discriminative test 2.0; App. 4), and each policy deducts the cost of an executed check from its remaining budget, stopping checking when the budget is exhausted or no admissible pair remains. Ordinary agent steps are never charged to B; they are reported separately as steps and dollar cost. All ten configurations share the same B and the same stopping conditions, so the comparison isolates where each policy spends its verification budget. Table 11 reports the realized consumption of every configuration.

Table 11: Realized verification-budget consumption per configuration (mean per instance over 300 instances $\times$ three seeds; oracles are reference runs).

Configuration	Checks executed	Check cost ($)	Budget used (% of B)	Rounds of re-ranking
No verification	0	0.000	0.0%	0
Random pair	1.2	0.0009	9%	1
Uncertainty ranking	1.8	0.0014	13%	1
Post-fail. reflection	2.1	0.0016	15%	1
State-level gate	2.6	0.0020	19%	1
Verify-every-write	14.7	0.0112	84%	1
Heuristic debt score	5.9	0.0044	41%	3.2
Debt-aware (ours)	6.3	0.0047	45%	3.5
Truth oracle	5.1	0.0038	36%	3.0
Value oracle	5.4	0.0040	39%	3.0

Table 12: Transfer of debt-aware selection across scaffolds and backbones. All rows use the backbone stated in the row label; "Codex CLI" and "OpenHands" refer to the agent scaffold run with the deepseek-chat backbone, *not* to the official products' reported scores.

Configuration	No verif.	Debt-aware	Δ (pp)
mini-swe-agent + deepseek-chat (main)	31.9%	48.8%	+16.9
OpenHands + deepseek-chat	29.0%	50.7%	+21.7
Codex CLI + deepseek-chat	42.0%	52.0%	+10.0
mini-swe-agent + deepseek-reasoner	38.7%	51.7%	+13.0
mini-swe-agent + claude-sonnet-4-5	48.7%	58.7%	+10.0

The point of the table is *where* the budget goes: verify-every-write spends on checks triggered by writes with little downstream exposure, while debt-aware selection concentrates the same budget on high-reach assumptions. This is what the cost–rework trade-off (Fig. 7) and the budget-sensitivity curve (Fig. 3b) show aggregately: identical B, very different realized value.

C.5 Budget Sensitivity

Fig. 3b varies the verification budget B (normalized check cost) around the default of 20. No verification does not consume the budget and is unaffected. Debt-aware selection shows saturating returns, and verify-every-write at $4\times$ its budget still trails debt-aware selection at $1\times$. Indiscriminate verification cannot buy back the resolution that selective verification gets for a fraction of the cost.

C.6 Cross-Scaffold and Cross-Backbone Transfer

Table 12 reports the no-verification and debt-aware resolution when the same controller is ported to two additional scaffolds (OpenHands, Codex CLI) and two additional backbones (deepseek-reasoner, claude-sonnet-4-5), all on the 300-instance benchmark (one seed for these runs). Because each transfer row is a single seed, we treat the results as directional evidence: the gain is positive in every configuration and it shrinks on stronger baselines (Codex CLI $+10.0$ pp, claude-sonnet-4-5 $+10.0$ pp) as the baseline absorbs more of the recoverable headroom; we do not attach confidence intervals to these rows.

C.7 Public Reference Points and Leaderboard Positioning

Table 13 positions the reported results against publicly reported systems on SWE-bench. Two caveats make the comparison asymmetric by design. First, every number in this table is an *absolute* system–model score from a different scaffold and budget; our contribution is the *within-scaffold causal estimate* of Sec. 4.2, where scaffold, backbone, tool interface, and stopping conditions are held fixed and only the verification policy varies. Second, the official SWE-bench Verified leaderboard (Jimenez et al., 2024) is evaluated with mini-SWE-agent as the harness, so its top entries are the closest public anchor to our scaffold; we list them without implying competitiveness on absolute resolution. The

Table 13: Public reference points on SWE-bench (reported by the cited sources; not evaluated by us) and our within-scaffold results. The Verified rows come from the official leaderboard (SWE-bench Team, 2024) (mini-SWE-agent harness, updated 2026-02); the Lite rows from the original papers and the official leaderboard; the three rows of this work use our 300-instance subset.

System (model)	Split	Resolved	Source
Public reference points (absolute scores)			
Claude 4.5 Opus (mini-SWE-agent)	Verified	76.8%	(SWE-bench Team, 2024)
Gemini 3 Flash (mini-SWE-agent)	Verified	75.8%	(SWE-bench Team, 2024)
Claude 4.6 Opus (mini-SWE-agent)	Verified	75.6%	(SWE-bench Team, 2024)
GPT-5.2 Codex (mini-SWE-agent)	Verified	72.8%	(SWE-bench Team, 2024)
Claude 4.5 Sonnet (mini-SWE-agent)	Verified	71.4%	(SWE-bench Team, 2024)
ExpeRepair (Claude-4 + o4-mini)	Lite	60.3%	Mu et al. (2025)
Globant Code Fixer Agent	Lite	48.3%	(SWE-bench Team, 2024)
OpenHands + CodeAct (claude-3.5-sonnet)	Lite	41.7%	Wang et al. (2024)
Agentless-1.5 (GPT-4o)	Lite	32.0%	(SWE-bench Team, 2024)
Agentless (GPT-4o)	Lite	27.3%	Xia et al. (2024)
SWE-agent (GPT-4)	Lite	18.0%	Xia et al. (2024)
This work (within-scaffold, deepseek-chat backbone)			
mini-swe-agent, no verification	Subset	31.9%	Table 1
mini-swe-agent, debt-aware (ours)	Subset	48.8%	Table 1
mini-swe-agent, claude-sonnet-4-5, debt-aware	Subset	58.7%	Table 12

verified split (500 instances) and our 300-instance subset are distinct and are not compared against each other.

Because the verified split is evaluated with the same mini-SWE-agent harness, the leaderboard entries are a meaningful upper reference for what the scaffold can reach with frontier models; our configuration deliberately uses the cheaper deepseek-chat backbone, so its absolute resolution is expected to sit below those entries. The within-scaffold gain ($+16.9$ pp on the 300-instance subset; $+10.0$ pp on the stronger claude-sonnet-4-5 backbone) is the causal claim of this paper, and it is orthogonal to the absolute level of the underlying backbone: the debt-aware controller adds value on every backbone and scaffold we ported it to (App. C.6).

C.8 Extended Check Types

Branch comparison, deferred from Sec. 3.2, runs two candidate implementations of the code region in question and compares their observable behavior. It is used only where a second candidate implementation is available at a live checkpoint; in the full run it accounted for 4.1% of executed checks and never changed a verdict relative to the discriminative test for the same hypothesis.

C.9 Construct-Validity Details

Table 14 gives the full construct-validity quantities behind Sec. 4.4. These rows answer different validity questions and are not one pooled estimator: F1 and κ are a reliability audit on three double-annotated held-out trajectories; the stated-assumption count and undecidable rate are over the ten pilot trajectories; behavioral redirection is observed on eight paired forks selected at live-assumption checkpoints. The paired forks are intentionally reported as descriptive mechanism audits: their outcomes are not used to fit the main-run confidence intervals, and selection-conditioned positive values do not estimate an average treatment effect.

C.10 Pilot Debt Statistics

Table 15 reports the manually annotated debt outcomes on the ten pilot trajectories referenced in Sec. 4.2: every hypothesis was labeled under the schema of Sec. 3.1, and "realizations" are assumptions whose later refutation forced rework of downstream actions.

Table 14: Construct-validity audits. F1 and κ are measured on three double-annotated held-out trajectories; the stated-assumption count and undecidable rate are over the ten pilot trajectories (26 assumptions); behavioral redirection and net value are descriptive outcomes on eight selection-conditioned paired forks, not a population treatment effect.

Quantity	Value
Claim agreement (F1)	0.824
Type Cohen's κ	0.774
Undecidable rate	7.7%
Behavior-change rate (8 forks)	0.75 (6/8)
Outcome-flip rate (8 forks)	0.50 (4/8)
Net-value-positive rate	1.0
Mean net value / mean check cost	\$0.105 / \$0.007

Table 15: Debt statistics on the ten pilot trajectories (manual annotation).

Quantity	No verification	Debt-aware (ours)
Assumptions formed	26	11
Unsupported at formation	12 (46.2%)	3 (27.3%)
Refuted later	11	1
Debt realizations	16	**0**
Wasted actions	138	**0**
Reverted diff lines	405	**0**
Lock-in events	10	**0**

C.11 Reach–Value Atlas

Fig. 8 visualizes the eight paired forks of Sec. 4.6: radial distance encodes avoidable downstream actions (reach), endpoint area encodes realized net verification value, and the endpoint style records whether the check changed downstream edits or only confirmed the current path. Inner glyphs identify the check type. The inset exposes the underlying pointwise relationship ($r{=}0.98$, $p{<}0.001$), which we report as descriptive only: the sample is eight pilot forks, the forks are selection-conditioned on the extractor reporting a live assumption, and reach and realized value both derive from the same downstream actions, so the correlation is partly mechanical (Sec. 4.6); all eight checks have positive net value.

C.12 Additional and Negative Results

Table 16 collects the results that did not come out in our favor: uncertainty and post-hoc-reflection policies do not significantly beat no verification; verify-every-write cannot buy resolution with budget; the gain is negligible on the symbolic-reasoning repositories; the learned scorer does not beat the heuristic on those held-out repositories; and the transfer gain shrinks on stronger scaffolds.

For reference, the 10-instance pilot (Sec. 4.2) used the same pipeline (five configurations, one seed, plus the paired-fork analysis of Sec. 4.4); its aggregate numbers are superseded by the full run. Its trajectories and annotations are in the supplementary material (`pilot_data/`) and fixed the annotation schema and fork protocol before the full run.

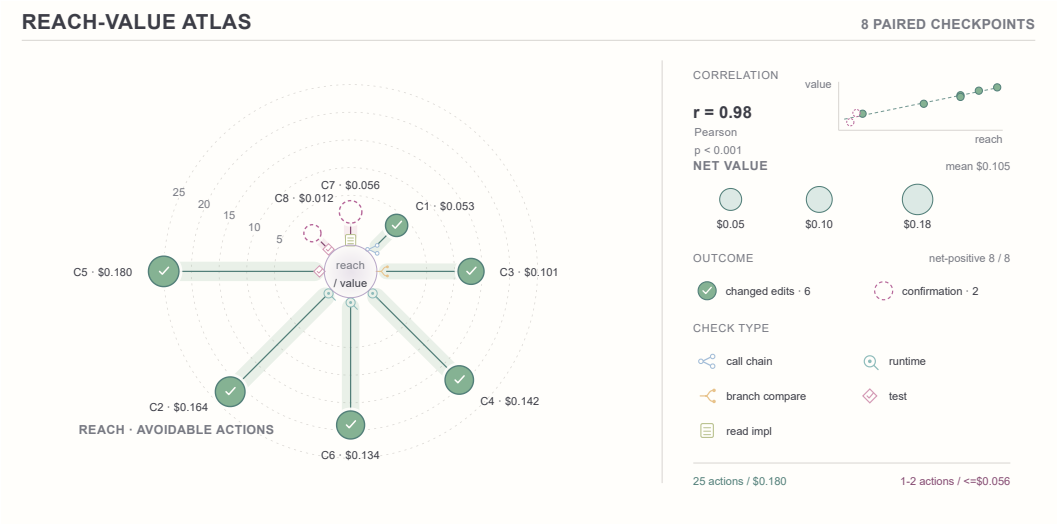

Figure 8: Reach–value atlas across eight selection-conditioned paired forks. The figure is descriptive: it is not a population estimate of verification effects, and reach and realized value share downstream-action units.

Table 16: Negative results. Deltas use instance-block bootstrap 95% CIs; a result is marked n.s. when the CI contains zero.

Result	Value
Uncertainty ranking vs. no verification	$+3.1$ pp, CI $[-0.8, +6.9]$ (n.s.)
Post-failure reflection vs. no verification	$+2.5$ pp, p=0.227 (n.s.)
State-level gate vs. no verification	$+2.8$ pp, p=0.166 (n.s.)
Verify-every-write at 4× budget vs. debt-aware at 1×	47.4% vs. 48.8%
Method gain on SymPy / xarray	$+4.8$ pp / -3.9 pp (n.s.)
Learned vs. heuristic on held-out SymPy / xarray	p=0.42 / p=0.51 (n.s.)
Transfer gain: mini-swe-agent vs. Codex CLI	$+16.9$ pp vs. $+10.0$ pp

C.13 CENSORING SENSITIVITY

Truncation (Sec. 4.1) understates the realized rework of a truncated branch, and the bias direction depends on *which* branch is truncated: truncating a control branch understates $L_{>t}^{\text{ctrl}}$ and therefore *understates* the realized value ($\widehat{V} = L_{>t}^{\text{ctrl}} - L_{>t}^{\text{intv}} - c_t(v)$, Eq. 7), while truncating an intervention branch understates $L_{>t}^{\text{intv}}$ and *overstates* it. We therefore adjust both directions separately, assigning every truncated control branch a minimum loss of one step-equivalent and, as a conservative bound, the mean rework of non-truncated control branches; truncated intervention branches receive the same minimum. These are lower-bound adjustments, not worst cases, so the analysis shows robustness to the *observed* truncation rather than to an unbounded loss. Table 17 reports the resolution and gain over no verification under each adjustment.

Table 17: Truncation sensitivity of the main result. Rows differ in which truncated branches are adjusted and by how much, using the truncation rates of Sec. 4.1; control truncation biases the realized value downward and intervention truncation upward.

Adjustment	Debt-aware	No verification	Gain (pp)
None (main table)	48.8%	31.9%	+16.9
Control branches ← +1 step-equivalent	48.7%	31.9%	+16.8
Intervention branches ← +1 step-equivalent	48.8%	31.9%	+16.9
Both ← +1 step-equivalent	48.7%	31.9%	+16.8
Control branches ← mean control rework	48.5%	31.9%	+16.6

D CODE AND REPRODUCIBILITY

All code and data behind the reported numbers are released with the camera-ready artifacts: full-run trajectories, official SWE-bench evaluation outputs, and trained V_θ weights. The strategy layer (graph, extraction, check generation, scoring, controller) is a self-contained package under `experiment/debtaware/`; reported numbers come from `code/audit_numbers.py` and `code/block_bootstrap.py` over `full_run_data/`, figures from `code/gen_figures.py`, and pilot annotations and paired-fork checkpoints from `pilot_data/`. Seeds, configurations, and API costs are in `full_run_data/meta.json`.