PROCCTRLBENCH: EVALUATING PROCESS-LEVEL DEFECTS AND CONTROL PRESERVATION IN LLM CODING AGENTS

Anonymous authors

ABSTRACT

Outcome-based evaluation of LLM coding agents provides limited information about defects that arise during execution. We present ProcCtrlBench, a benchmark for execution-process evaluation in LLM coding agents. ProcCtrlBench organizes recurrent execution defects into a reusable ontology covering 11 defect types in 4 categories, and evaluates agent trajectories through standardized process evidence rather than final outcomes alone. To support cross-platform comparison, ProcCtrlBench standardizes raw logs into a shared trajectory representation and reports calibrated scorecards over process-level findings. In addition, ProcCtrlBench uses control preservation to quantify execution-process quality, capturing whether execution remains interpretable, interruptible, correctable, reversible, and able to hand back authority when needed. We evaluate ProcCtrlBench on 200 execution trajectories sampled from three benchmarks, corresponding to 2,200 defect-level evaluation points under the full 11-defect annotation protocol. Results show that ProcCtrlBench can be instantiated with useful reliability, provides more stable semantics than direct thresholding, and reveals distinct differences in execution quality often overlooked by conventional outcome-based metrics. Our code are available at `https://anonymous.4open.science/r/ProcCtrlBench-655C`.

1 INTRODUCTION

Existing benchmarks for LLM coding agents primarily evaluate final outcomes, such as task completion, compilation success, and test pass rates (Jimenez et al., 2024; Merrill et al., 2026; Chen et al., 2021). These metrics (Bean et al., 2026; Wallach et al., 2025) are useful for measuring overall capability, but they provide limited visibility into how an agent executes a multi-step task (Kim et al., 2026). In realistic coding settings, an agent may eventually solve a task while following a trajectory that is repetitive, inefficient, weakly structured, or difficult to supervise (Kwa et al., 2025). As coding agents become more autonomous, this gap between final-outcome evaluation and execution-process quality becomes increasingly important (Schwartz et al., 2026).

However, many practically important failures arise during execution rather than only at the endpoint (Salaudeen et al., 2025; Ibrahim et al., 2025). As a result, two runs with similar final outcomes may still differ substantially in execution quality, diagnosability, and practical usability (Liao & Wortman Vaughan, 2024). This limitation of endpoint-based evaluation is increasingly recognized beyond coding agents. ProcBench (Fujisawa et al., 2024), for example, directly evaluates multi-step inference instead of relying only on final-answer accuracy. TRACE (Wang et al., 2023) likewise shows that, under continual learning, preserving reasoning-related behavior is important because adaptation can degrade general capability and instruction following. Together, these works support our view that coding-agent evaluation should consider not only whether a task is solved, but also how reliably the execution process is maintained.

To address this limitation, we present ProcCtrlBench, a benchmark for process evaluation in LLM coding agents. ProcCtrlBench organizes recurrent defects into a reusable ontology covering 11 defect types in four categories, evaluates agent trajectories through process evidence rather than final outcomes, and standardizes raw logs into a unified trajectory representation to support comparison across heterogeneous agents. It further uses control preservation to quantify execution-process quality, capturing whether execution remains interpretable, interruptible, correctable, reversible, and able to hand back authority when needed. ProcCtrlBench reports calibrated scorecards

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.

over process-level findings, including defect-level findings, dimension-level process-quality scores, control-preservation scores, and a compact summary score for coarse comparison. ProcCtrlBench complements existing outcome metrics by adding a process-level diagnostic layer.

We evaluate ProcCtrlBench on 200 execution trajectories sampled from three benchmarks: AndroidBench (AndroidBench Contributors, 2026), Terminal-Bench (Merrill et al., 2026), and SWE-bench-Verified (Jimenez et al., 2024). Because each trajectory is annotated against all 11 defect classes, this study is supported by 2,200 defect-level judgments in addition to trajectory-level system comparisons. Our results show that ProcCtrlBench can be instantiated with useful reliability, provides more stable semantics than direct thresholding, and reveals meaningful differences in execution quality that are often overlooked by conventional outcome-based evaluation.

Our contributions are summarized as follows:

- We propose ProcCtrlBench, a benchmark for execution-process evaluation in LLM coding agents beyond final outcomes.
- We introduce a reusable ontology of execution defects, covering 11 defect types in four categories, to support structured benchmarking of agent trajectories.
- We standardize heterogeneous execution logs into a unified trajectory representation and produce calibrated scorecards over process-level findings, with control preservation used to quantify execution-process quality.
- We evaluate ProcCtrlBench on 200 cases sampled from AndroidBench, Terminal-Bench, and SWE-bench-Verified, and show that it can be instantiated with useful reliability and identifies process-level differences invisible to outcome-only metrics.

2 Related Work

Research on LLM evaluation has mainly emphasized outcome-oriented benchmarks (Kiela et al., 2021; Srivastava et al., 2023; Eriksson et al., 2025; Bean et al., 2026), which measure whether a system succeeds but provide limited visibility into how it gets there (Schwartz et al., 2026). This limitation is especially important for multi-step agents, where failures often emerge during execution, such as context degradation, repetitive tool use, inefficient action chains, and loss of control (Cemri et al., 2026; Weidinger et al., 2023). Recent work has therefore moved toward process-sensitive evaluation. ProcBench (Fujisawa et al., 2024) evaluates whether models follow explicit procedures, and TRACE (Wang et al., 2023) highlights the importance of preserving reasoning-related behavior. These works provide motivation, but do not directly benchmark coding-agent execution trajectories. ProcCtrlBench instead targets recurring process defects and trace-based control proxies.

Recent agent-evaluation studies have also examined intermediate behavior, including planning traces, tool-use trajectories, error recovery, and failure analysis in realistic environments (Liu et al., 2024; Zhu et al., 2026; Weidinger et al., 2023; Liao & Wortman Vaughan, 2024; Schwartz et al., 2026). These studies demonstrate that passing final tests does not guarantee stable intermediate execution. Related work also includes qualitative and task-specific analyses (Ibrahim et al., 2025; Berman et al., 2024) and fine-grained progress-based evaluation such as AgentBoard (Ma et al., 2024). ProcCtrlBench complements this line of work by organizing recurring coding-agent execution defects into a shared taxonomy and diagnostic scorecard.

Research on uncertainty estimation and calibration (He et al., 2026; Shorinwa et al., 2025; Huang et al., 2024) improves the interpretability of model outputs through confidence estimation (Detommaso et al., 2024; Lin et al., 2024), disagreement-based methods (Xiong et al., 2024), and post hoc calibration (Ulmer et al., 2024), but these methods are usually applied to single-turn responses or final predictions. ProcCtrlBench instead estimates probabilities for process-level findings derived from sequential execution trajectories.

Work on human oversight and control in autonomous systems (Weidinger et al., 2023; Flournoy et al., 2020; Manheim & Homewood, 2025; Matias, 2023) further emphasizes that reliable automation depends not only on success, but also on whether behavior remains understandable, interruptible, correctable, and safe to hand back. These concerns are especially important for long-horizon coding agents (Schwartz et al., 2026). ProcCtrlBench uses them to organize trace-based control proxies alongside defect findings.

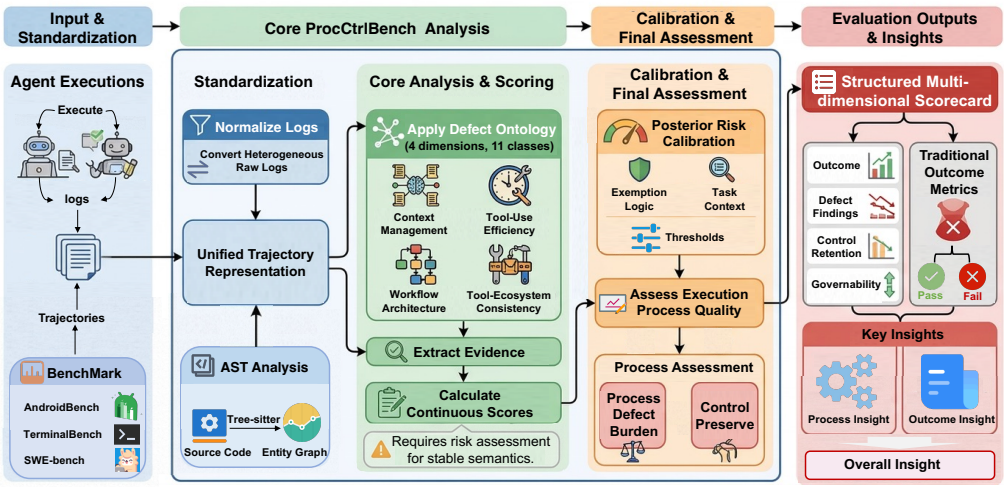


Figure 1: Overview of ProcCtrlBench. The framework transforms platform-specific agent trajectories into calibrated process scorecards.

3 METHODS

3.1 PROCCTRLBENCH OVERVIEW

Figure 1 summarizes the design of ProcCtrlBench. The central challenge is that different coding agents expose heterogeneous execution logs, tool traces, and workflow structures, which makes direct process comparison difficult (Wang et al., 2025; Zhu et al., 2026). ProcCtrlBench addresses this by first transforming raw executions into a standardized trajectory representation, so that different systems can be evaluated within a common process space. On top of this unified representation, ProcCtrlBench applies a reusable ontology of execution defects covering 11 defect types in four categories. Because process defects exist on a continuum rather than as binary events, ProcCtrlBench decouples evidence extraction, risk estimation, and scorecard reporting.

The architecture maps detector evidence to estimated defect-occurrence probabilities, with task context and exemption logic as intended inputs. Calibration is intended to improve the interpretation of defect-occurrence probabilities within the evaluated setting. Finally, ProcCtrlBench aggregates defect findings into a structured scorecard that reports process-defect burden, dimension-level quality scores, control preservation, and a compact overall summary. In our presentation, the full scorecard is the primary output; the scalar summary is intended only for coarse comparison.

3.2 STANDARDIZED TRAJECTORY REPRESENTATION

ProcCtrlBench compares heterogeneous agent systems through a standardized trajectory representation rather than raw platform-specific logs. Coding agents differ in logging formats, tool interfaces, message schemas, and workflow structures; without standardization, process-level findings would be difficult to compare across systems.

Given a raw execution log L, we apply a deterministic mapping

$$\Psi : L \to T,$$

where

$$T = (e_1, e_2, \ldots, e_n)$$

is an ordered sequence of standardized events. Each event is represented as

$$e_i = (t_i, m_i, u_i, v_i, o_i, c_i, d_i),$$

where t_i denotes event type, m_i the textual payload, u_i the tool invocation, v_i the tool return or validation result, o_i the observable external operation, c_i the context state, and d_i the structural dependency information, including the parent step, branch identity, workflow unit, or agent identity.

This layer is not itself a benchmark dimension. Rather, it serves as a common interface that enables downstream evidence extraction. In the current implementation, defect evidence may be derived from count-based features, similarity patterns, dependency-graph structure, and context-state transitions, depending on defect type.

3.3 Defect Taxonomy and Benchmark Instantiation

ProcCtrlBench structures process evaluation around a reusable defect ontology. Let Z denote the set of defect types benchmarked by ProcCtrlBench, with each $z_j \in Z$ corresponding to a specific execution defect. In this work, ProcCtrlBench covers eleven defect types organized into four top-level categories: Context Management, Tool-Use Efficiency, Workflow Architecture, and Tool-Ecosystem Consistency. For each defect class, ProcCtrlBench extracts structured evidence

$$x_j = \phi_j(T),$$

and then computes a continuous evidence score,

$$s_j = f_j(x_j).$$

We retain s_j as a continuous signal because many process defects are graded rather than naturally binary. When binary activation is needed, it is derived only as an intermediate interpretation layer:

$$\text{finding}_j = \begin{cases} 1, & s_j \geq \tau_j \wedge \neg \text{exempt}_j(T), \\ 0, & \text{otherwise}, \end{cases}$$

Exemption logic is important because some trajectories may superficially match a defect pattern while remaining semantically appropriate under task context, such as intended validation reruns or necessary long-lived instruction text.

A key design choice is that the ontology and the current instantiation should not be conflated. The ontology defines what kinds of failures ProcCtrlBench aims to evaluate; the detectors, thresholds, and scoring rules used here are one practical instantiation of that benchmark space.

3.4 Calibrated Process Scoring

Raw detector scores do not by themselves provide consistent semantics across defect types or task conditions. The same score magnitude may correspond to different empirical defect likelihoods under different execution contexts (D'Amour et al., 2022). ProcCtrlBench proposes probability-based interpretation of benchmark evidence before scorecard reporting.

For the j-th defect, let $y_j \in \{0, 1\}$ denote the adjudicated presence label under the annotation protocol, x_j the extracted evidence, and c the task and environment context. The target defect-occurrence probability is

$$p_j = P(y_j = 1 \mid x_j, c).$$

For interpretability, posterior risks are mapped into semantic bands using thresholds $\delta_w < \delta_e$:

$$\text{band}_j = \begin{cases} \text{error}, & p_j \geq \delta_e, \\ \text{warning}, & \delta_w \leq p_j < \delta_e, \\ \text{none}, & p_j < \delta_w. \end{cases}$$

The names "warning" and "error" identify lower and higher defect-likelihood bands, not levels of harm or consequence severity. Likewise, "none" denotes a probability below δ_w, not proof of defect absence. ProcCtrlBench aggregates defect-level risks into dimension-level process-quality scores across the four benchmark dimensions, and then into an overall process-quality summary Q_{def}. Because the benchmark is intended primarily as a structured reporting tool, we emphasize the full scorecard over any single scalar.

3.5 Control Preservation and Benchmark Output

ProcCtrlBench uses control preservation to quantify execution-process quality during autonomous execution (Flournoy et al., 2020). This matters because a trajectory may reach a correct outcome while still being difficult to interpret, interrupt, repair, or safely halt.

We decompose control preservation into five subdimensions: interpretability, interruptibility, correctability, reversibility, and authority handoff. Together, these properties capture whether execution remains legible to an external observer, whether intervention is possible at meaningful points, whether local deviations can be repaired without restarting the trajectory, whether the system can return to a prior safe state, and whether decision authority can be returned when needed.

Let $\psi(T)$ denote control-related features extracted from trajectory T, and let F denote the set of benchmark findings. We define the aggregate control-preservation score as

$$CP = g(\psi(T), F).$$

A low defect count does not guarantee high control preservation. For example, an agent might avoid all defined defects by executing a single, opaque, monolithic script that passes tests but cannot be interrupted or rolled back. We therefore measure CP independently of aggregate defect burden.

For compact comparison, ProcCtrlBench reports a summary score

$$PB = \eta Q_{\text{def}} + (1 - \eta)CP, \quad 0 \leq \eta \leq 1.$$

However, we stress that this scalar is secondary. The primary benchmark output is the full scorecard, which exposes defect-specific findings, dimension-level quality, control-preservation assessments, and coverage information.

Although ProcCtrlBench is proposed primarily as a benchmark, its calibrated process findings may also support execution-time governance. In this setting, benchmark findings and current control state can be used to regulate autonomy, trigger confirmation or rollback, or request handoff. We emphasize that this downstream use is not part of the benchmark definition itself; it is included only to illustrate that benchmark-calibrated process signals may also be operationally useful.

4 Experiments

We evaluate ProcCtrlBench through four questions: (1) whether the benchmark can be instantiated reliably from real execution trajectories; (2) whether calibrated scoring provides more stable benchmark semantics than direct thresholding; (3) whether ProcCtrlBench reveals system-level differences beyond outcome-based metrics; and (4) whether the benchmark remains informative across task scenarios, with exploratory analysis of auxiliary downstream utility.

4.1 Experimental Setup

We evaluate ProcCtrlBench on 200 execution trajectories sampled from SWE-bench-Verified (100), AndroidBench (40), and Terminal-Bench (60), spanning 11 agent–model configurations from four agent families. Under the annotation protocol, these trajectories produce 2,200 defect-level judgments across 11 process-defect classes. We use trajectories as the unit for system-level comparison and defect-level judgments as the supervision scale for defect instantiation. Each trajectory is independently reviewed by two annotators, with third-reviewer adjudication when needed; further annotation details and agreement statistics are provided in Appendix B.

The annotated subset is split into 80 development cases, 40 calibration cases, and 80 held-out evaluation cases, stratified by benchmark source and task success. All available execution logs are transformed into standardized trajectories before scoring.

We report precision, recall, F1, Average Precision, and AUROC for defect instantiation, ECE for calibration quality, and PB, dimension-level quality scores, CP, and fragile-success rate for system comparison. Implementation details are provided in Appendix B.3.

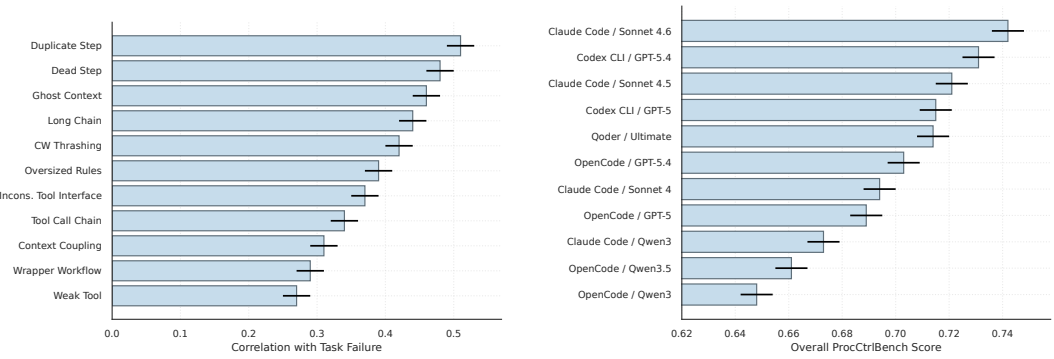
4.2 RQ1: Instantiation Reliability

We first evaluate whether ProcCtrlBench defect classes can be instantiated with useful reliability from real execution trajectories. For this analysis, the 200 annotated trajectories yield 2,200 defect-level judgments under the full annotation protocol, with effective sample size varying by defect due to prevalence and exemption. This is a prerequisite for meaningful benchmark use: if benchmarked defects cannot be grounded in trajectory evidence with reasonable consistency, downstream comparison becomes weakly supported.

Table 1: Defect-level detection and calibration summaries for the 11 benchmarked process defects on the annotated set. Higher is better for detection metrics; lower is better for ECE.

Defect-level Instantiation								Calibration (ECE)	
Group	Defect Benchmark	Trigger	Precision	Recall	F1	AP	AUROC	Hard	Bayes
Context Management	Ghost Context	**0.61**	**0.84**	**0.86**	**0.85**	0.83	0.91	0.214	0.118
	Oversized Rules	0.43	0.81	0.79	0.80	0.79	0.88		
	CW Thrashing	0.47	0.78	0.80	0.79	0.77	0.86		
Tool-Use Efficiency	Duplicate Step	0.38	0.86	0.85	**0.85**	**0.85**	**0.92**	**0.198**	**0.103**
	Tool Call Chain	0.29	0.79	0.76	0.77	0.75	0.85		
	Dead Step	0.42	0.83	0.81	0.82	0.81	0.90		
	Long Chain	0.55	0.80	0.83	0.81	0.80	0.89		
Workflow Architecture	Wrapper Workflow	0.21	0.61	0.57	0.59	0.56	0.71	0.271	0.196
	Context Coupling	0.26	0.64	0.59	0.61	0.58	0.73		
Tool Ecosystem	Incons. Tool Interface	0.24	0.76	0.74	0.75	0.73	0.84	0.223	0.134
	Weak Tool	0.17	0.55	0.52	0.53	0.50	0.68		
Overall	-	-	-	-	-	-	-	0.227	0.138

CW Thrashing: Context Window Thrashing. ECE: Expected Calibration Error.



(a) Correlation between defect benchmarks and downstream task failure

(b) Overall ProcCtrlBench score across agent–model configurations

Figure 2: Overview results for ProcCtrlBench. The left panel shows the association between defect benchmarks and downstream task failure. Long Chain and Dead Step show the strongest associations, while Context Window Thrashing, Context Coupling, Duplicate Step, and Ghost Context also exhibit meaningful positive correlations. The right panel shows the overall ProcCtrlBench score across the 11 evaluated agent–model configurations.

Table 1 reports precision, recall, F1, AP, and AUROC for all 11 defect classes on the annotated set. Defects with strong local observability, such as Ghost Context, Duplicate Step, Dead Step, and Long Chain, achieve stronger performance. Structurally higher-level defects such as Wrapper Workflow, Context Coupling, and Weak Tool are more difficult to instantiate, which is expected because they depend more heavily on latent workflow structure and higher-level execution intent. Figure 2 (left) shows associations between each defect and downstream task failure. Long Chain and Dead Step show the strongest positive correlations, followed by Context Window Thrashing and Context Coupling. These patterns indicate that ineffective local progress, elongated execution, and degraded context handling directly contribute to execution breakdown.

Overall, the results suggest that several ProcCtrlBench defect classes can be instantiated with useful reliability, although observability varies by defect type and should be interpreted accordingly.

4.3 RQ2: Calibration Quality

We next evaluate whether calibrated scoring improves the empirical interpretability of ProcCtrlBench findings relative to direct thresholding. Table 1 reports Expected Calibration Error for direct hard-threshold interpretation and hierarchical Bayesian calibration. Across all four defect dimensions, calibrated posterior risk yields lower ECE. The gains are largest in Workflow Architecture and

Tool-Ecosystem Consistency, where evidence is more heterogeneous and observability is weaker. In these settings, a single hard threshold is especially brittle because the same detector magnitude may correspond to different empirical defect likelihoods under different conditions.

These results suggest that posterior-risk-based scoring provides more consistent empirical interpretation than raw trigger decisions in our annotated setting. Additional analyses indicate that warning- and error-level findings correspond to increasingly higher empirical defect frequency, supporting the interpretability of the severity bands; reliability diagrams in Appendix C.8 further show that calibrated risks more closely track empirical defect frequency than hard-threshold outputs. At the same time, these conclusions should be read in light of the current annotation scale and the lack of broader out-of-distribution validation.

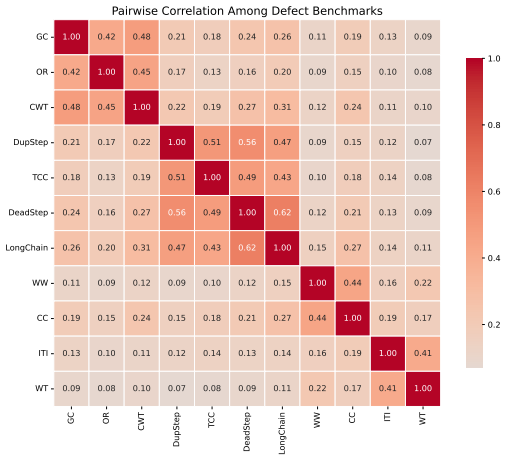


Figure 3: Pairwise correlation matrix among defect-level posterior risks on the mixed evaluation set. Moderate correlations appear within some efficiency-related defects, while many cross-dimension pairs remain weaker.

We also examine the dependence structure among defect-level posterior risks to assess whether ProcCtrlBench collapses into a largely redundant signal. Figure 3 shows that some efficiency-related defects, especially Long Chain and Dead Step, exhibit moderate positive association, which is expected because elongated execution often co-occurs with locally ineffective steps. However, correlations across dimensions are generally weaker, suggesting that ProcCtrlBench does not reduce to a single latent defect factor. This supports the scorecard design, in which multiple partially related process defects are reported together rather than summarized as a raw defect count.

4.4 RQ3: Cross-System Benchmarking Beyond Outcome Metrics

The main purpose of ProcCtrlBench is to compare coding-agent systems beyond final-outcome performance. We therefore evaluate all 11 agent–model configurations using the full ProcCtrlBench scorecard. Table 2 reports the overall ProcCtrlBench score, dimension-level quality scores, control preservation, and fragile-success rate. The right panel of Figure 2 summarizes the overall ranking, and Figure 4 provides per-defect profiles. Several patterns emerge.

First, ProcCtrlBench exposes meaningful differences in process profile across systems. Claude Code with Claude Sonnet 4.6 achieves the highest overall summary score, while Codex CLI is particularly competitive in tool-use efficiency. OpenCode variants remain competitive in some outcome-oriented settings but are weaker in workflow quality, tool-ecosystem consistency, control preservation, and fragile-success rate. Second, process quality appears to depend jointly on model capability and agent orchestration. Within the same framework, stronger model variants generally improve the summary score and reduce fragile success. Across frameworks, similar model families behave differently once embedded in different agent infrastructures. This supports the view that process reliability is a property of the combined agent–model system rather than of the base model alone. Third, ProcCtrlBench mainly adds diagnostic granularity beyond conventional outcome metrics. In most cases it does not radically overturn the overall ranking, but it helps distinguish systems with similar endpoint performance

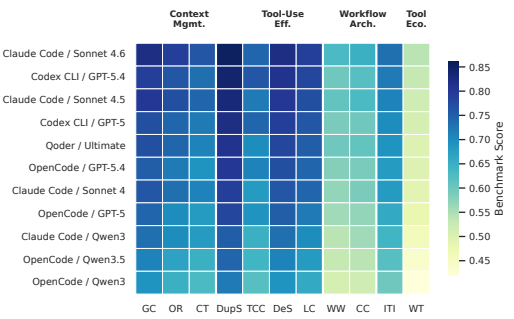


Figure 4: Reported per-defect profiles for the 11 configurations. Rows denote systems; columns denote the eleven defects in Appendix A.

Table 2: Cross-agent and cross-model comparison under ProcCtrlBench on the 200-trajectory mixed evaluation set. The left block reports overall scorecard metrics, and the right block reports the overall ProcCtrlBench score PB on each task source. Higher is better for PB, Q_{ctx}, Q_{tool}, Q_{wf}, Q_{eco}, and CP; lower is better for Fragile Success.

Agent	Model	Overall Scorecard							Scenario-specific PB		
		PB	Q_{ctx}	Q_{tool}	Q_{wf}	Q_{eco}	CP	Fragile Succ.	Android	Terminal	SWE-bench
Claude Code	Claude Sonnet 4.6	**0.742**	**0.78**	0.74	**0.71**	0.69	**0.75**	**10.8%**	**0.756**	0.731	**0.744**
	Claude Sonnet 4.5	0.721	0.76	0.71	0.69	0.67	0.73	12.1%	0.733	0.716	0.724
	Claude Sonnet 4	0.694	0.72	0.69	0.66	0.65	0.70	15.4%	0.704	0.689	0.697
	Qwen3 Coder Plus	0.673	0.70	0.66	0.64	0.63	0.68	17.3%	0.682	0.664	0.671
Codex CLI	gpt-5.4-0305-global	0.731	0.74	**0.75**	0.68	**0.70**	0.72	11.7%	0.721	**0.742**	0.732
	GPT-5 Global	0.715	0.72	0.74	0.67	0.68	0.71	12.9%	0.707	0.728	0.717
Open Code	gpt-5.4-0305-global	0.703	0.71	0.72	0.65	0.67	0.69	14.6%	0.694	0.711	0.704
	GPT-5 Global	0.689	0.69	0.71	0.64	0.66	0.68	15.8%	0.681	0.697	0.691
	qwen3.5-plus	0.661	0.67	0.65	0.63	0.61	0.66	18.5%	0.654	0.672	0.664
	Qwen3 Coder Plus	0.648	0.65	0.64	0.61	0.60	0.65	20.2%	0.641	0.659	0.651
Qoder	Ultimate mode	0.714	0.73	0.70	0.69	0.66	0.71	13.6%	0.708	0.719	0.716

Q_{ctx}: Context Mgmt.; Q_{tool}: Tool-Use Eff.; Q_{wf}: Workflow Arch.; Q_{eco}: Tool Ecosystem.
Android: AndroidBench; Terminal: Terminal-Bench; SWE-bench: SWE-bench-Verified.

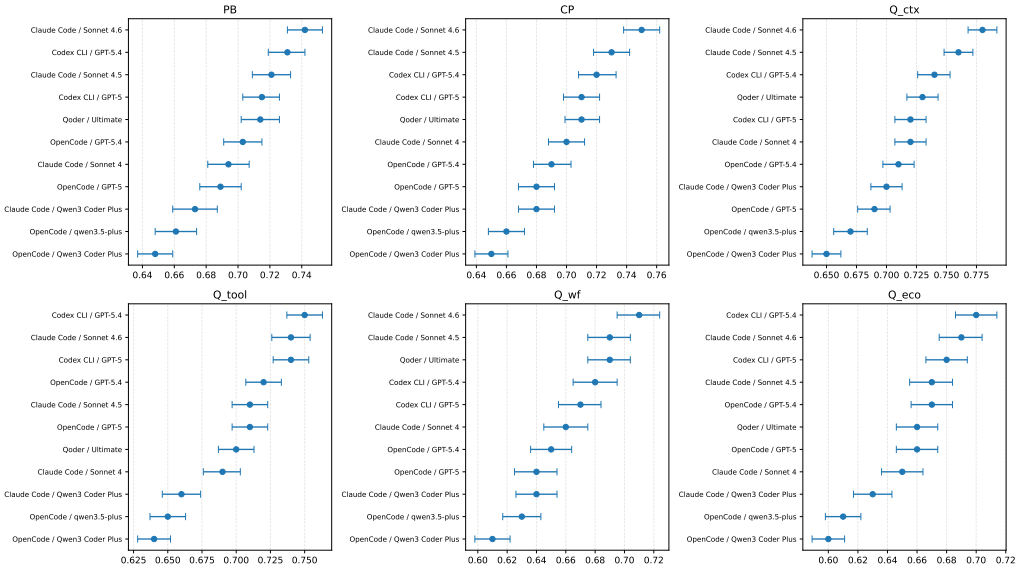


Figure 5: Bootstrap confidence intervals for overall ProcCtrlBench score, control preservation, and dimension-level process-quality scores across the 11 evaluated systems. Error bars indicate 95% bootstrap confidence intervals.

by revealing differences in process-level findings, workflow quality, and control preservation. This is particularly visible in systems whose outcome scores remain competitive while their trajectories exhibit weaker process structure or higher fragile-success rates.

These results suggest that ProcCtrlBench can support structured comparative evaluation beyond end-point success alone, especially when users care about reliability, diagnosability, and governability in addition to task completion. We further assess the stability of these comparisons under trajectory resampling. Figure 5 reports bootstrap confidence intervals for the overall ProcCtrlBench score, control preservation, and dimension-level process-quality scores across the 11 evaluated systems. The results suggest that coarse-grained distinctions between stronger and weaker systems are stable, while several middle-ranked systems remain statistically close. This indicates that ProcCtrlBench is informative for structured comparative evaluation, although small differences between near-tied systems should be interpreted with caution.

We further compare ProcCtrlBench against an outcome-only baseline to quantify the diagnostic gain (details in Appendix C.5). The comparison shows that ProcCtrlBench usually preserves the coarse ordering of stronger and weaker systems, but adds finer diagnostic separation in cases where endpoint performance is similar but execution quality differs. This reinforces the intended role of ProcCtrlBench as a complementary process-oriented evaluation layer rather than a wholesale replacement for outcome-based metrics.

4.5 RQ4: Scenario Sensitivity and Generality

We examine whether ProcCtrlBench remains informative across task scenarios that emphasize different execution demands. Table 2 reports scenario-specific ProcCtrlBench scores on AndroidBench, Terminal-Bench, and SWE-bench-Verified; Appendix Table 6 provides the same breakdown in a compact benchmark-by-benchmark view. The results show that system strengths vary meaningfully by benchmark source: Claude Code with Sonnet 4.6 performs best on AndroidBench and SWE-bench-Verified, while Codex CLI with GPT-5.4 performs best on Terminal-Bench. Rankings shift across benchmarks because AndroidBench stresses workflow structure, Terminal-Bench favors tool-centric interaction, and SWE-bench-Verified targets repository-level debugging. Despite these scenario-specific shifts, top-tier systems (e.g., Claude Code / Sonnet 4.6) maintain their global advantage due to consistently lower fragile-success rates. This suggests that ProcCtrlBench captures both scenario-sensitive behavior and a broader notion of execution-process quality, rather than collapsing to source-specific noise or an over-smoothed aggregate.

Additional analyses in Appendix C.8 and Appendix C.3 further probe the stability and interpretability of benchmark findings. In particular, removing calibration weakens the empirical consistency of finding severity, while removing exemption logic increases false positives for defects such as Duplicate Step and Ghost Context. These analyses support the role of calibration and exemption handling in maintaining interpretable process-level findings. Overall, the current results suggest that ProcCtrlBench generalizes reasonably across the three evaluated coding scenarios while preserving sensitivity to the execution properties that distinguish them.

5 Limitations

ProcCtrlBench has several limitations. First, its ontology is not exhaustive: the 11 defect classes introduced here are intended as a practical starting point for process-oriented benchmarking rather than a complete taxonomy of coding-agent failures. Second, some defect classes are only partially observable from execution traces, especially higher-level structural defects such as Wrapper Workflow, Context Coupling, and Weak Tool. Third, the current instantiation depends on process-level annotation, and although we now report annotation workflow, agreement, and data-splitting details, the scale of this annotation remains limited relative to the breadth of agent behaviors that may arise in practice. Fourth, calibrated risk estimation is based on a relatively small annotated set and may be sensitive to distribution shifts across task styles, tool ecosystems, and workflow regimes. Fifth, control preservation should be understood as an operational proxy for governability rather than a complete theory of human oversight. More broadly, the current benchmark should be interpreted as a structured measurement framework for process-level diagnostics, not as a definitive measure of overall agent capability, safety, or deployability. Future work should validate its conclusions under larger-scale annotations, broader logging regimes, and real intervention settings before using ProcCtrlBench as evidence for high-stakes system selection or governance decisions.

6 Conclusion

We presented ProcCtrlBench, a benchmark for evaluating LLM coding agents beyond final outcomes. By organizing recurring execution defects into a reusable ontology and assessing standardized trajectories with calibrated process-level scorecards, ProcCtrlBench provides a structured view of execution quality, including control preservation during agent behavior. Experiments on 200 annotated cases from AndroidBench, Terminal-Bench, and SWE-bench-Verified suggest that several defect types can be instantiated with useful reliability and that calibrated process assessment offers diagnostic value beyond outcome-based evaluation alone. More broadly, our findings suggest that evaluating coding agents should consider not only whether tasks are completed, but also whether the underlying execution process remains reliable, interpretable, and governable.

AI USE STATEMENT

In this work, we used generative AI tools for drafting and editing parts of the manuscript to improve clarity and readability, and for editing software code to help identify and correct implementation defects. We have not used generative AI tools to generate synthetic data, to produce annotations or benchmark labels, to design the core methodology, or to determine or interpret the final experimental results. We reviewed all AI-assisted work. All AI-assisted text was carefully checked and revised by the authors, and all AI-assisted code was reviewed, tested, and validated by the authors before use. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

AndroidBench Contributors. AndroidBench: A framework for benchmarking LLMs on Android development tasks. GitHub repository, 2026. URL `https://github.com/android-bench/android-bench`.

Andrew M Bean, Ryan Othniel Kearns, Angelika Romanou, Franziska Sofia Hafner, Harry Mayne, Jan Batzner, Negar Foroutan Eghlidi, Chris Schmitz, Karolina Korgul, Hunar Batra, et al. Measuring what matters: Construct validity in large language model benchmarks. *Advances in Neural Information Processing Systems*, 38, 2026.

Glen Berman, Nitesh Goyal, and Michael Madaio. A scoping study of evaluation practices for responsible ai tools: Steps towards effectiveness evaluations. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, pp. 1–24, 2024.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, et al. Why do multi-agent llm systems fail? *Advances in Neural Information Processing Systems*, 38, 2026.

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 e-prints*, pp. arXiv–2107, 2021.

Alexander D'Amour, Katherine Heller, Dan Moldovan, Ben Adlam, Babak Alipanahi, Alex Beutel, Christina Chen, Jonathan Deaton, Jacob Eisenstein, Matthew D Hoffman, et al. Underspecification presents challenges for credibility in modern machine learning. *Journal of Machine Learning Research*, 23(226):1–61, 2022.

Gianluca Detommaso, Martin Bertran, Riccardo Fogliato, and Aaron Roth. Multicalibration for confidence scoring in llms. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 10624–10641, 2024.

Maria Eriksson, Erasmo Purificato, Arman Noroozian, Joao Vinagre, Guillaume Chaslot, Emilia Gomez, and David Fernandez-Llorca. Can we trust ai benchmarks? an interdisciplinary review of current issues in ai evaluation. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, volume 8, pp. 850–864, 2025.

Michèle A Flournoy, Avril Haines, and Gabrielle Chefitz. Building trust through testing: Adapting dod's test & evaluation, validation & verification (tevv) enterprise for machine learning systems, including deep learning systems. *WestExec Advisors, Washington DC*, 2020.

Ippei Fujisawa, Sensho Nobe, Hiroki Seto, Rina Onda, Yoshiaki Uchida, Hiroki Ikoma, Pei-Chun Chien, and Ryota Kanai. Procbench: Benchmark for multi-step reasoning and following procedure. *arXiv preprint arXiv:2410.03117*, 2024.

Wenchong He, Zhe Jiang, Tingsong Xiao, Zelin Xu, and Yukun Li. A survey on uncertainty quantification methods for deep learning. *ACM Computing Surveys*, 58(7):1–35, 2026.

Hsiu-Yuan Huang, Yutong Yang, Zhaoxi Zhang, Sanwoo Lee, and Yunfang Wu. A survey of uncertainty estimation in llms: Theory meets practice. *arXiv e-prints*, pp. arXiv–2410, 2024.

Lujain Ibrahim, Saffron Huang, Lama Ahmad, Umang Bhatt, and Markus Anderljung. Towards interactive evaluations for interaction harms in human-ai systems. In *Proceedings of the AAAI/ACM Conference on AI, Ethics, and Society*, volume 8, pp. 1302–1310, 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.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, Zhengxuan Wu, Bertie Vidgen, Grusha Prasad, Amanpreet Singh, Pratik Ringshia, et al. Dynabench: Rethinking benchmarking in nlp. In *Proceedings of the 2021 conference of the North American chapter of the Association for Computational Linguistics: human language technologies*, pp. 4110–4124, 2021.

Jeonghye Kim, Minseon Kim, Young Jin Kim, Matheus Pereira, Marc-Alexandre Côté, Alessandro Sordoni, Xingdi Yuan, and Zhengyan Shi. Programdistill: From interactive web apps to verifiable reference-guided swe tasks. *arXiv preprint arXiv:2609.18805*, 2026.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, et al. Measuring ai ability to complete long tasks. *arXiv e-prints*, pp. arXiv–2503, 2025.

Q Vera Liao and Jennifer Wortman Vaughan. Ai transparency in the age of llms: A human-centered research roadmap. *Harvard Data Science Review*, 5, 2024.

Zhen Lin, Shubhendu Trivedi, and Jimeng Sun. Generating with confidence: Uncertainty quantification for black-box large language models. *Transactions on Machine Learning Research*, 2024, 2024.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. In *International Conference on Learning Representations*, volume 2024, pp. 52989–53046, 2024.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents, 2024. URL https://arxiv.org/abs/2401.13178.

David Manheim and Aidan Homewood. Limits of safe ai deployment: Differentiating oversight and control. *arXiv e-prints*, pp. arXiv–2507, 2025.

J Nathan Matias. Humans and algorithms work together—so study them together. *Nature*, 617 (7960):248–251, 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 e-prints*, pp. arXiv–2601, 2026.

Olawale Elijah Salaudeen, Anka Reuel, Ahmed M Ahmed, Suhana Bedi, Zachary Robertson, Sudharsan Sundar, Benjamin W Domingue, Angelina Wang, and Sanmi Koyejo. Measurement to meaning: A validity-centered framework for ai evaluation. In *NeurIPS 2025 Workshop on Evaluating the Evolving LLM Lifecycle: Benchmarks, Emergent Abilities, and Scaling*, 2025.

Reva Schwartz, Carina Westling, Morgan Briggs, Marzieh Fadaee, Isar Nejadgholi, Matthew Holmes, Fariza Rashid, Maya Carlyle, Afaf Taïk, Kyra Wilson, et al. Circle: A framework for evaluating ai from a real-world lens. *arXiv e-prints*, pp. arXiv–2602, 2026.

Ola Shorinwa, Zhiting Mei, Justin Lidard, Allen Z Ren, and Anirudha Majumdar. A survey on uncertainty quantification of large language models: Taxonomy, open research challenges, and future directions. *ACM Computing Surveys*, 58(3):1–38, 2025.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *Transactions on machine learning research*, 2023.

Dennis Ulmer, Martin Gubri, Hwaran Lee, Sangdoo Yun, and Seong Oh. Calibrating large language models using their generations only. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15440–15459, 2024.

Hanna Wallach, Meera Desai, A Feder Cooper, Angelina Wang, Chad Atalla, Solon Barocas, Su Lin Blodgett, Alexandra Chouldechova, Emily Corvi, P Alex Dow, et al. Position: Evaluating generative ai systems is a social science measurement challenge. *arXiv e-prints*, pp. arXiv–2502, 2025.

Xiao Wang, Yuansen Zhang, Tianze Chen, Songyang Gao, Senjie Jin, Xianjun Yang, Zhiheng Xi, Rui Zheng, Yicheng Zou, Tao Gui, et al. Trace: A comprehensive benchmark for continual learning in large language models. *arXiv preprint arXiv:2310.06762*, 2023.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

Laura Weidinger, Maribeth Rauh, Nahema Marchal, Arianna Manzini, Lisa Anne Hendricks, Juan Mateos-Garcia, Stevie Bergman, Jackie Kay, Conor Griffin, Ben Bariach, et al. Sociotechnical safety evaluation of generative ai systems. *arXiv e-prints*, pp. arXiv–2310, 2023.

Miao Xiong, Zhiyuan Hu, Xinyang Lu, Yifei Li, Jie Fu, Junxian He, and Bryan Hooi. Can llms express their uncertainty. *An empirical evaluation of confidence elicitation in LLMs. arXiv*, 2306, 2024.

Yuxuan Zhu, Tengjun Jin, Yada Pruksachatkun, Andy Zhang, Shu Liu, Sasha Cui, Sayash Kapoor, Shayne Longpre, Kevin Meng, Rebecca Weiss, et al. Establishing best practices in building rigorous agentic benchmarks. *Advances in Neural Information Processing Systems*, 38, 2026.

A Detailed Benchmark Definitions

This appendix expands the definitions and boundary conditions for the process defects introduced in the main text. ProcCtrlBench is designed to benchmark recurring execution pathologies that plausibly arise in realistic coding-agent deployment, rather than assigning arbitrary labels to trajectories after the fact. In long-horizon coding tasks, many failures emerge not from a single catastrophic mistake, but from the interaction between context pressure, tool interfaces, workflow composition, and execution control. The benchmark definitions below therefore emphasize three aspects: what the defect is, where its practical boundary lies, and why it matters for process-level evaluation.

A.1 Context Management

This dimension concerns the preservation, saturation, and degradation of contextual resources during execution. In coding-agent settings, context is a constrained operational resource that must support task continuity, tool coordination, and error recovery over multiple steps.

Ghost Context. Ghost Context refers to redundant, outdated, or already summarized content that remains in context for an extended period and continues to consume the token budget. Let u denote a context segment, with occupancy ratio $\rho(u)$, downstream reference rate $\kappa(u)$, and persistence duration $\tau(u)$. This defect typically appears as high occupancy, low reference, and long persistence. Necessary long-lived rule text, explicitly retained summaries, and persistent memory are excluded.

In practice, Ghost Context often arises when agents retain raw file excerpts, command outputs, or earlier observations after their informational role has already been superseded by a concise summary or a more recent state. The risk is not merely wasted tokens, but also increased anchoring on stale evidence and reduced room for more relevant context.

Oversized Rules. Oversized Rules refers to excessively large system prompts, policy text, or instruction blocks that impose a persistent context burden at every step. Unlike Ghost Context, this defect concerns static rule overhead rather than dynamically retained redundant content.

This defect often appears in deployments where safety rules, style guidelines, workflow instructions, and tool documentation are accumulated into a single large prompt prefix. Even when each component is individually reasonable, the combined overhead can consume a substantial fraction of the context window and reduce effective space for task-specific reasoning.

Context Window Thrashing. Context Window Thrashing denotes a high-frequency pattern of heavy information loading followed by rapid saturation and short-term compression, in which information is compressed or discarded before it can form a stable reasoning chain.

The core issue is not large context alone, but unstable context turnover. Agents repeatedly acquire, compress, and replace information without preserving enough connective structure for durable reasoning. This often leads to rediscovery behavior, weak hypothesis continuity, and repeated local reconstruction of facts that had already been observed earlier in the trajectory.

A.2 Tool-Use Efficiency

This dimension concerns repeated calls, local looping, ineffective steps, and disproportionately long execution chains in tool use. In deployed coding agents, tool interaction is the primary interface between model reasoning and environment action, so failures in this dimension manifest directly as repeated actions yielding no state change.

Duplicate Step. Duplicate Step refers to the repeated execution of highly similar tool calls despite the absence of substantive state change. Validation reruns, time-varying commands, and intended batch processing are exempt.

This defect commonly arises when the agent lacks sufficient evidence to decide whether a previous call resolved its uncertainty, and therefore repeats low-information actions such as reopening the same file span, rerunning near-identical searches, or issuing repeated status checks without meaningful environmental change. The main signal is weak local progress rather than repetition alone.

Tool Call Chain. Tool Call Chain captures cyclic or oscillatory patterns in local call sequences, such as repeated retries of the same tool or loops of the form $A \to B \to A \to B$.

Its practical significance lies in local entrapment: the agent remains active, but its tool sequence no longer reflects strategic escalation or replanning. Compared with Duplicate Step, Tool Call Chain emphasizes the structure of repeated local interaction rather than similarity of individual calls alone.

Dead Step. Dead Step describes a step whose action is executed but whose result neither enters later messages nor affects later tool inputs, file modifications, branch decisions, or state updates.

Dead Steps indicate weak causal integration within the trajectory. Tool outputs may be retrieved, inspected, or generated, but if they exert no downstream influence, the step becomes operationally inert. This matters because it reflects not only inefficiency, but also poor coupling between observation and subsequent action.

Long Chain. Long Chain concerns whether the overall execution path is disproportionately long. Its core signal is not the step count itself, but persistent elongation in the absence of effective task decomposition, stage control, and checkpointing.

Long trajectories are not inherently defective, especially in repository-level debugging or multi-stage validation tasks. ProcCtrlBench therefore treats Long Chain as a structured burden signal rather than a length threshold alone. It is activated when execution keeps expanding without corresponding consolidation of progress.

A.3 Workflow Architecture

This dimension evaluates whether workflow structure provides real benefit. Modern coding agents increasingly use skills, wrappers, staged pipelines, and sub-agent decomposition, but such structure helps only when it improves validation, routing, fault tolerance, or control.

Wrapper Workflow. Wrapper Workflow refers to a workflow or skill that consistently performs only one-step pass-through behavior without adding value through validation, aggregation, routing, or fault tolerance.

The defect does not target modularity itself. Rather, it targets architectural indirection that increases complexity while contributing little operational benefit. Thin wrappers may still be acceptable when they enforce policy or standardize interfaces, but repeated pass-through behavior without clear functional gain is benchmarked as structural overhead.

Context Coupling. Context Coupling refers to excessively deep context sharing and unclear boundaries across agents or workflow units, often manifested as bidirectional dependencies, ping-pong alternation, or strongly connected components.

Some degree of context sharing is expected in coordinated systems. The defect applies when sharing becomes so deep that unit boundaries lose meaning, local reasoning becomes difficult to isolate, and errors propagate across workflow components. This reduces diagnosability and weakens the ability to correct local deviations without broader trajectory disruption.

A.4 Tool-Ecosystem Consistency

This dimension concerns inconsistency within the surrounding tool ecosystem. Execution quality depends not only on model capability and workflow logic, but also on whether tools are discoverable, distinguishable, and interface-consistent for model-driven use.

Inconsistent Tool Interface. Inconsistent Tool Interface describes substantial inconsistency among functionally similar tools in parameter naming, parameter types, output structure, or error-return formats.

This defect is especially common in organically grown tool ecosystems. For LLM agents, such inconsistencies are more than minor usability issues: they make invocation patterns harder to generalize, increase malformed calls, and encourage fallback to less suitable but more familiar alternatives.

Weak Tool. Weak Tool refers to a tool that exists but is difficult for the model to discover and use reliably, for example when its invocation rate remains persistently low in clearly appropriate settings and it is often replaced by a more intuitive but only partially overlapping alternative.

The main issue is not capability absence, but ecosystem legibility. A tool may be technically available yet underused because its name, description, or interface fails to make its function salient to the model. In such cases, the benchmark captures weakness of exposure and practical accessibility rather than intrinsic tool quality alone.

B Experimental Setup Details

B.1 Annotation Protocol

To validate benchmark instantiation, we annotate a subset of 200 trajectories sampled from the full corpus using reviewers with prior experience in AI coding or agent-based tool use. Each trajectory is assessed at the process level rather than solely at the endpoint. Annotators judge whether each targeted defect is present, absent, or exempt under task context, and record brief evidence notes together with exemption rationales where needed. Because the current benchmark instantiation covers 11 defect classes, full annotation over 200 trajectories yields 2,200 defect-level judgments in total (200×11), although these judgments do not imply 2,200 observed defects because many labels are absent or exempt.

The protocol is process-oriented rather than outcome-oriented. Annotators are instructed not to label a pattern as defective from surface form alone. For example, repeated validation is not marked as Duplicate Step when intervening edits justify reruns, and persistent rule text is not marked as Ghost Context when it still serves coordination or control. This matters because many process defects are only weakly reflected in final task success or failure.

All trajectories are independently reviewed by two annotators. When the two annotators disagree on the presence, absence, or exemption status of a defect, the case is sent to a third reviewer for adjudication. In addition to binary judgments, annotators record short evidence notes to preserve the local semantic context needed for process-level review.

B.2 Dataset and Annotation Statistics

The reported study uses 200 annotated cases drawn from three benchmark sources: 100 from SWE-bench-Verified, 40 from AndroidBench, and 60 from Terminal-Bench. These cases are selected to cover a range of coding scenarios, including repository-level bug fixing, terminal-centric tool use, and Android-oriented engineering tasks. The evaluation includes 11 agent–model configurations spanning four agent families: Claude Code, Codex CLI, OpenCode, and Qoder.

We use the following terminology throughout the paper. A *task* denotes a benchmark problem, a *run* denotes one system execution on that task, and a *trajectory* denotes the recorded event sequence for that run. The annotated subset is designed to include short-, medium-, and long-horizon executions, as well as both tool-light and tool-rich environments, so that context-, tool-, workflow-, and ecosystem-related defects are all represented.

We report two complementary notions of evaluation scale. The study contains 200 annotated execution trajectories, which serve as the primary unit for system-level benchmarking and score aggregation. Under the full annotation protocol, each trajectory is labeled against all 11 defect classes, yielding 2,200 defect-level judgments in total. Table 3 summarizes this accounting by benchmark dimension. These judgments should not be interpreted as observed defect occurrences, because each judgment may be present, absent, or exempt, and the effective sample size remains defect-dependent.

This diversity is important because process defects do not arise uniformly across task types. Compact terminal tasks more clearly expose local redundancy and inefficient tool use, while longer software-engineering tasks more often surface context accumulation, structural drift, and workflow coupling. Sampling across the three sources therefore helps reduce the risk that ProcCtrlBench overfits to a single execution regime.

Table 3: Defect-level evaluation-point accounting under the full annotation protocol. Counts are computed assuming complete labeling of all 11 defect classes for each of the 200 annotated trajectories. These are annotation judgments rather than observed defect occurrences.

Benchmark Dimension	Defect Classes per Trajectory	Total over 200 Trajectories
Context Management	3	600
Tool-Use Efficiency	4	800
Workflow Architecture	2	400
Tool Ecosystem Consistency	2	400
Total	11	2,200

Table 4: Conventional outcome metrics and ProcCtrlBench scorecard side by side across the 11 evaluated systems. Rank Shift is computed as Outcome Rank minus ProcCtrlBench Rank; positive values indicate that a system ranks better under ProcCtrlBench than under conventional outcome metrics.

System	Outcome Score	Outcome Rank	PB	PB Rank	Rank Shift	Fragile Success
Claude Code / Sonnet 4.6	0.768	1	0.742	1	0	10.8%
Codex CLI / GPT-5.4	0.759	2	0.731	2	0	11.7%
Claude Code / Sonnet 4.5	0.752	3	0.721	3	0	12.1%
Codex CLI / GPT-5	0.748	4	0.715	4	0	12.9%
Qoder / Ultimate	0.744	6	0.714	5	1	13.6%
OpenCode / GPT-5.4	0.745	5	0.703	6	-1	14.6%
Claude Code / Sonnet 4	0.733	7	0.694	7	0	15.4%
OpenCode / GPT-5	0.731	8	0.689	8	0	15.8%
Claude Code / Qwen3 Coder Plus	0.724	9	0.673	9	0	17.3%
OpenCode / Qwen3.5	0.719	10	0.661	10	0	18.5%
OpenCode / Qwen3 Coder Plus	0.712	11	0.648	11	0	20.2%

Outcome Score can be instantiated as task-success rate, test-pass rate, or a normalized outcome composite depending on logging availability.

B.3 Hyperparameter and Implementation Notes

All available execution logs are first transformed by Ψ into standardized trajectories before evidence extraction and scoring. ProcCtrlBench then produces per-defect findings and estimated occurrence probabilities, dimension-level process-quality scores, control-preservation assessments, and overall benchmark scores.

Detector thresholds are initialized using heuristic rules and pilot inspection on the development split, and are then refined against annotated trajectories to balance precision and recall. Exemption rules are applied after score computation and before posterior risk estimation. This ordering preserves raw detector evidence while reducing false positives for contextually appropriate behavior.

Posterior-risk estimation uses different practical implementations depending on detector family. Low-dimensional count-based detectors use simple Bayesian updates or frequency-smoothed empirical estimation, while higher-dimensional detectors use calibrated score-to-risk mappings fit on held-out annotation data. Shared priors may be used within defect families when beneficial, with class-specific adjustment retained for defect classes that differ sufficiently evidence distributions.

For warning and error grading, the thresholds δ_w and δ_e are selected on the calibration split to preserve monotonic separation among low-, medium-, and high-risk findings on the annotated set. These implementation choices are intended to support stable empirical interpretation within the evaluated setting rather than to define a universally fixed operating point.

B.4 Annotation Agreement and Data Splits

To improve transparency of benchmark instantiation, we report the annotation workflow, agreement statistics, and data partition used in the current study. Each trajectory in the 200-trajectory annotated subset is independently labeled by two annotators, with disagreements resolved by a third reviewer

Table 5: Annotation agreement before adjudication on the 200-trajectory annotated subset. Agreement is reported at the defect level; under the full protocol this corresponds to 2,200 defect-level judgments in total, not 2,200 defect occurrences.

Defect	Raw Agreement	Cohen's κ	Exemption Rate	Adjudication Rate
Ghost Context	0.87	0.73	0.06	0.12
Oversized Rules	0.84	0.68	0.04	0.15
Context Window Thrashing	0.82	0.64	0.05	0.17
Duplicate Step	0.89	0.77	0.08	0.10
Tool Call Chain	0.83	0.66	0.07	0.16
Dead Step	0.86	0.72	0.05	0.13
Long Chain	0.88	0.75	0.03	0.11
Wrapper Workflow	0.76	0.52	0.05	0.22
Context Coupling	0.78	0.55	0.04	0.21
Inconsistent Tool Interface	0.81	0.61	0.03	0.18
Weak Tool	0.74	0.47	0.06	0.24
Overall	0.83	0.65	0.05	0.17

Table 6: Scenario-specific ProcCtrlBench scores across benchmark sources. Systems are sorted by overall mixed-set ProcCtrlBench score PB.

System	Overall	Android	Terminal	SWE-bench
Claude Code / Sonnet 4.6	**0.742**	**0.756**	0.731	**0.744**
Codex CLI / GPT-5.4	0.731	0.721	**0.742**	0.732
Claude Code / Sonnet 4.5	0.721	0.733	0.716	0.724
Codex CLI / GPT-5	0.715	0.707	0.728	0.717
Qoder / Ultimate	0.714	0.708	0.719	0.716
OpenCode / GPT-5.4	0.703	0.694	0.711	0.704
Claude Code / Sonnet 4	0.694	0.704	0.689	0.697
OpenCode / GPT-5	0.689	0.681	0.697	0.691
Claude Code / Qwen3 Coder Plus	0.673	0.682	0.664	0.671
OpenCode / Qwen3.5	0.661	0.654	0.672	0.664
OpenCode / Qwen3 Coder Plus	0.648	0.641	0.659	0.651

through adjudication. Agreement is computed before adjudication at the defect level; Table 5 reports the resulting raw agreement, Cohen's κ, exemption rate, and adjudication rate.

The annotated subset is split into 80 development cases, 40 calibration cases, and 80 held-out evaluation cases. The split is stratified by benchmark source and by endpoint task success or failure so that detector refinement, calibration, and final reporting are not dominated by a single task regime.

Agreement varies by defect type. Locally observable defects such as Duplicate Step and Long Chain are more consistent, while structurally higher-level defects such as Wrapper Workflow and Weak Tool are less consistent. This pattern matches the relative detector difficulty reported in Table 1.

C Additional Results

C.1 Scenario-Specific Scores

Table 6 reports the scenario-specific ProcCtrlBench scores for all evaluated systems on AndroidBench, Terminal-Bench, and SWE-bench-Verified. The results show that process quality is not monolithic across task regimes. Claude Code with Sonnet 4.6 remains strongest overall and performs best on AndroidBench and SWE-bench-Verified, while Codex CLI with GPT-5 achieves the

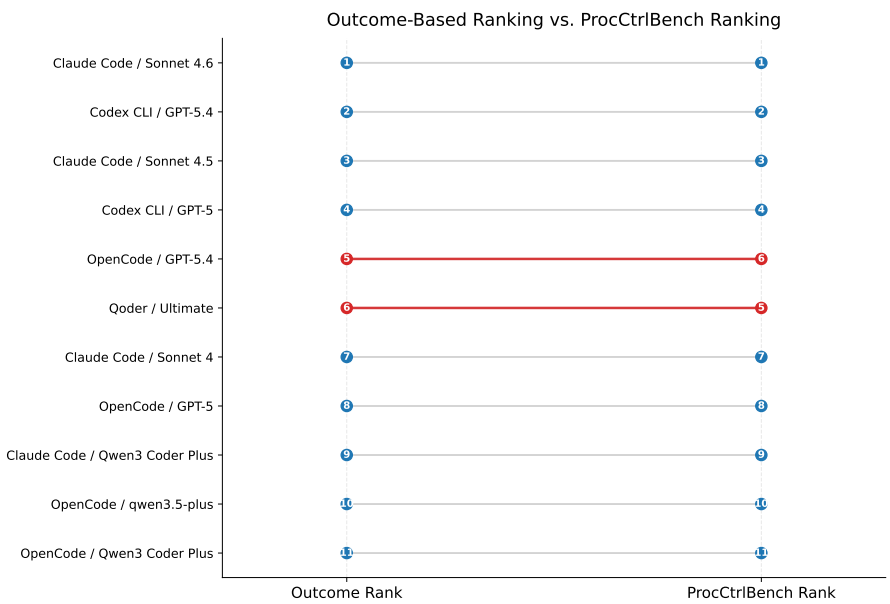


Figure 6: Comparison of conventional outcome-based ranking and ProcCtrlBench ranking across the 11 evaluated systems. Lines indicate rank shifts from outcome-based evaluation to ProcCtrlBench.

highest score on Terminal-Bench. This pattern suggests that different agent–model systems exhibit different strengths under different execution environments.

At the same time, the overall ordering is not completely unstable across sources. Systems that rank strongly under the mixed evaluation set generally remain competitive across individual benchmarks, although the margin of advantage varies by scenario. This indicates that ProcCtrlBench captures both global process quality and scenario-specific sensitivity, rather than collapsing to either benchmark-specific noise or an over-smoothed average.

C.2 Calibration Semantics

Additional analyses indicate that the warning and error bands carry meaningful empirical semantics: error-level findings correspond to the highest empirical defect frequency, warning-level findings occupy an intermediate regime, and non-triggered findings correspond to the lowest. This supports the use of semantic severity labels as an interpretable reporting layer over calibrated risk.

C.3 Exemption Rules and Auxiliary Utility

Additional analyses suggest that exemption rules are important for reducing false positives, particularly for Duplicate Step and Ghost Context. We also observe that ProcCtrlBench findings may have auxiliary downstream utility, including uncertainty-guided execution in high-risk trajectories.

C.4 Outcome-Based Ranking vs. ProcCtrlBench Ranking

Table 5 shows that agreement is highest for locally observable defects and lowest for structurally semantic defects, mirroring the relative instantiation difficulty observed in Table 1.

To clarify how ProcCtrlBench differs from conventional evaluation, we compare system ranking under outcome-based metrics and under ProcCtrlBench. Figure 6 visualizes rank shifts across the 11 evaluated systems, and Table 4 reports the corresponding scores and fragile-success rates.

The comparison shows that ProcCtrlBench does not merely rescale outcome performance. Some systems with similar final success differ materially in process quality and control preservation, while some systems with competitive outcome scores still exhibit elevated fragile-success rates. These results support the claim that ProcCtrlBench captures execution properties that are weakly reflected in conventional outcome-only benchmarks.

Table 7: Bootstrap ranking stability of ProcCtrlBench across the 11 evaluated systems on the mixed evaluation set. Values are computed over stratified bootstrap replicates.

System	Mean Rank	Rank Std	Top-1 Freq.	Top-3 Freq.
Claude Code / Sonnet 4.6	1.42	0.68	0.63	0.98
Codex CLI / GPT-5.4	2.11	0.81	0.22	0.94
Claude Code / Sonnet 4.5	2.87	0.93	0.10	0.81
Codex CLI / GPT-5	4.26	1.17	0.03	0.31
Qoder / Ultimate	4.39	1.12	0.02	0.28
OpenCode / GPT-5.4	5.92	1.35	0.00	0.07
Claude Code / Sonnet 4	6.41	1.28	0.00	0.03
OpenCode / GPT-5	7.18	1.41	0.00	0.01
Claude Code / Qwen3 Coder Plus	8.84	1.22	0.00	0.00
OpenCode / Qwen3.5	9.74	0.88	0.00	0.00
OpenCode / Qwen3 Coder Plus	10.86	0.42	0.00	0.00

C.5 Comparison to a Simpler Outcome-Only Baseline

A natural baseline is to rank systems using only endpoint outcome metrics, such as task success or test-pass rate. Table 4 compares this baseline with ProcCtrlBench. The comparison shows that ProcCtrlBench does not radically reorder the entire leaderboard; instead, it mainly adds diagnostic granularity by separating systems with similar outcome scores but different process profiles. This behavior is desirable for a process-oriented benchmark: the goal is not to replace endpoint evaluation, but to identify execution differences that endpoint metrics alone do not expose.

We summarize this effect in two ways. First, ProcCtrlBench preserves the top-3 systems under outcome-only ranking, indicating that stronger endpoint performance remains broadly aligned with stronger process quality at the coarse level. Second, ProcCtrlBench penalizes systems with elevated fragile-success rates and weaker workflow/control characteristics, even when their outcome scores remain competitive. OpenCode / GPT-5.4 is the clearest example in our current setting.

C.6 Bootstrap Stability Analysis

We assess the stability of ProcCtrlBench under trajectory resampling using stratified bootstrap over the 200-trajectory mixed evaluation set. The main text reports bootstrap confidence intervals for the overall ProcCtrlBench score, control preservation, and dimension-level process-quality scores, while Table 7 summarizes mean rank, rank standard deviation, Top-1 frequency, and Top-3 frequency across bootstrap replicates.

The results suggest that coarse-grained distinctions between stronger and weaker systems are stable, while several middle-ranked systems remain statistically close. This indicates that ProcCtrlBench is informative for structured comparative evaluation, but small differences between near-tied systems should be interpreted with caution.

C.7 Inter-Defect Correlation Structure

To examine whether ProcCtrlBench defects are excessively redundant, we compute pairwise correlations among defect-level posterior risks across the mixed evaluation set. The main text presents the resulting correlation matrix and highlights the main pattern.

Several efficiency-related defects, especially Long Chain and Dead Step, exhibit moderate positive association, as expected because elongated execution often co-occurs with locally ineffective steps. However, correlations across dimensions are generally weaker, suggesting that ProcCtrlBench does not collapse into a single latent defect signal. This supports the scorecard design in which multiple partially related process defects are reported together rather than summarized as a raw defect count.

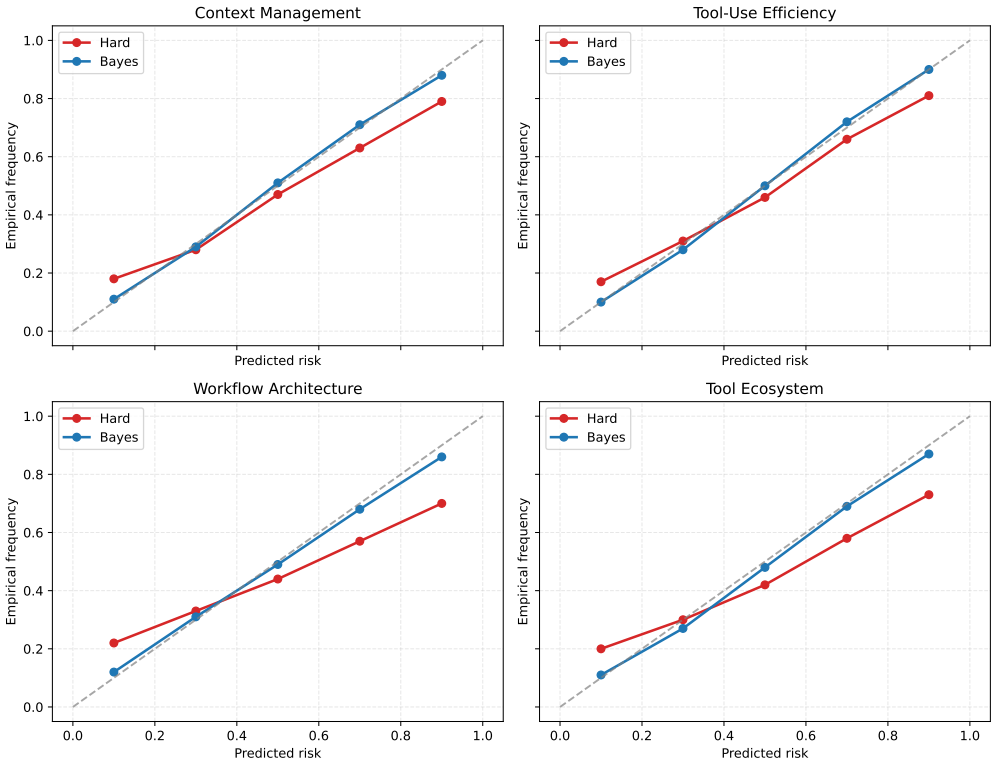


Figure 7: Reliability diagrams for hard-threshold interpretation and calibrated posterior risk across the four benchmark dimensions. Calibrated risks more closely track empirical defect frequency, especially in dimensions with weaker observability.

C.8 Calibration Reliability Diagrams

In addition to ECE, we visualize the relationship between predicted risk and empirical defect frequency using reliability diagrams. Figure 7 compares direct hard-threshold interpretation with calibrated posterior risk for the four benchmark dimensions.

The calibrated curves more closely follow the diagonal reference line and preserve monotonic separation between low-, medium-, and high-risk regions. The improvement is particularly visible in Workflow Architecture and Tool-Ecosystem Consistency, where benchmark evidence is more heterogeneous. These observations complement the quantitative ECE results reported in Table 1.

C.9 Sensitivity to the Weighting of Process Quality and Control Preservation

The overall ProcCtrlBench score combines aggregate process quality and control preservation through the weighting parameter η. To examine whether system comparison is overly sensitive to this design choice, we vary η from 0 to 1 and recompute system-level PB scores. Figure 8 reports the resulting score trajectories.

The strongest systems remain competitive across a broad range of η, while some middle-ranked systems exchange relative position depending on the relative weighting of aggregate process quality and control preservation. This result is consistent with our intended use of ProcCtrlBench: PB provides a useful summary, but the full scorecard remains the primary interpretation surface.

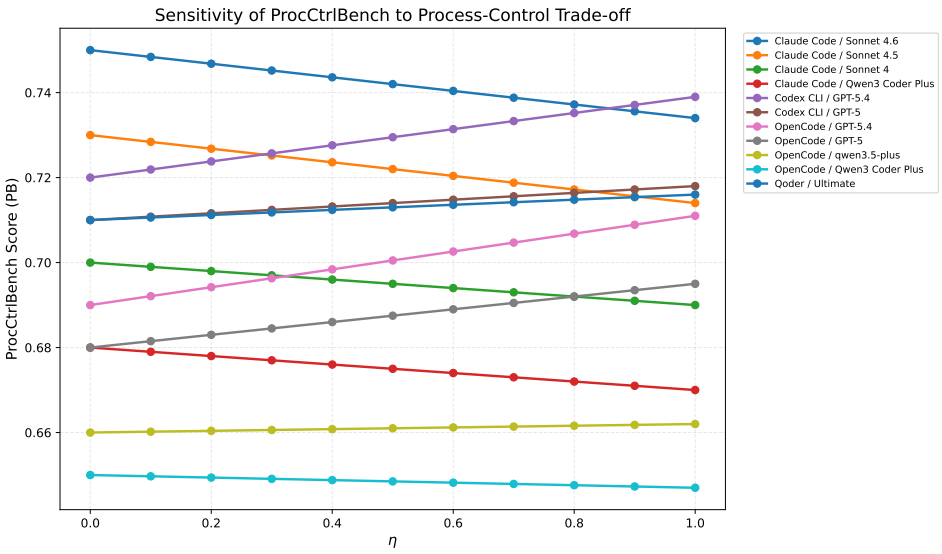


Figure 8: Sensitivity of system-level ProcCtrlBench scores to the trade-off parameter η between aggregate process quality and control preservation. Stronger systems remain competitive across a broad range of η, while some middle-ranked systems exchange relative order.

Table 8: Control-preservation subdimension scores across the 11 evaluated systems. Higher is better.

System	Interpretability	Interruptibility	Correctability	Reversibility	Authority Handoff	CP
Claude Code / Sonnet 4.6	0.77	0.74	0.76	0.73	0.75	0.75
Claude Code / Sonnet 4.5	0.75	0.72	0.74	0.71	0.73	0.73
Claude Code / Sonnet 4	0.72	0.69	0.71	0.68	0.70	0.70
Claude Code / Qwen3 Coder Plus	0.70	0.66	0.68	0.67	0.68	0.68
Codex CLI / GPT-5.4	0.74	0.72	0.73	0.70	0.72	0.72
Codex CLI / GPT-5	0.73	0.70	0.72	0.69	0.71	0.71
OpenCode / GPT-5.4	0.70	0.68	0.69	0.67	0.69	0.69
OpenCode / GPT-5	0.69	0.67	0.68	0.66	0.68	0.68
OpenCode / qwen3.5-plus	0.67	0.65	0.66	0.65	0.66	0.66
OpenCode / Qwen3 Coder Plus	0.66	0.64	0.65	0.64	0.65	0.65
Qoder / Ultimate	0.72	0.70	0.71	0.69	0.71	0.71

C.10 Control-Preservation Subdimension Breakdown

To improve transparency of control-preservation assessment, Table 8 reports subdimension-level scores for Interpretability, Interruptibility, Correctability, Reversibility, and Authority Handoff across the 11 evaluated systems.

The breakdown shows that control preservation is not simply the inverse of aggregate defect burden. Some systems maintain relatively strong interpretability and interruptibility despite weaker process efficiency, while others exhibit stronger process quality but less balanced control characteristics. This further supports using control preservation as an important component in quantifying execution-process quality, rather than treating it as a simple derivative of process-defect counts.