LEGO-UX: UNLOCKING THE VALUE OF HUMAN-AGENT INTERACTION TRAJECTORIES

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances in Large Language Models (LLMs) have substantially improved coding agents on single-turn benchmarks such as SWE-bench. However, real-world coding assistance is naturally interactive: users refine requirements, report failures, and correct unsuccessful attempts across turns. In this setting, user experience (UX) encompasses both test-verified task completion and interaction quality, including how effectively the agent follows and adapts to user feedback. Improving this experience calls for learning from real-world human–agent trajectories, which provide valuable supervision for multi-turn interactions but are difficult to leverage reliably. Their quality varies widely, complicating training data selection, while their underlying tasks often lack reproducible environments for verification and replay. We introduce LEGO-UX, a unified framework that uses interaction-aware trajectory selection and evidence-bounded task reconstruction for training and multi-turn evaluation. It identifies reliable supervision using behavioral and execution signals, while recovering executable tasks that facilitate multi-turn evaluation and controlled training rollouts. Using the reconstruction pipeline, we construct LEGO-UXBENCH, a 95-task benchmark for multi-turn agent user-experience. In the experiments, we evaluate 11 frontier models on LEGO-UXBENCH and observe that similar task-completion performance can correspond to markedly different interaction quality. Fine-tuning Qwen3.5-35B-A3B on selected trajectories improves both measured dimensions and raises the unweighted eight-metric macro score across four benchmarks by 15.2 points (7.75 to 22.91).

1 INTRODUCTION

Coding agents have made substantial progress in solving complex programming tasks (Jimenez et al., 2024; Deng et al., 2025). Yet real-world coding assistance extends beyond solving a fixed task from a single specification. Users interact with agents by refining requirements, reporting failures, and requesting revisions based on intermediate results. Effective coding agents must therefore adapt to evolving user needs throughout the interaction. Improving coding-agent user experience requires not only successful task completion, but also effective interaction with users throughout the process.

Real-world human–agent trajectories provide a natural resource for improving agent user experience (UX), capturing how users refine requirements and provide feedback, and how agents respond and adapt. Recent work has collected large-scale coding trajectories (Baumann et al., 2026), developed interactive benchmarks with persistent state and evolving requirements (Shen et al., 2026; Raghavendra et al., 2026; Wu et al., 2026b), and reconstructed executable tasks from agent trajectories for evaluation (Wu et al., 2026a). However, how to reliably leverage heterogeneous and often incomplete real-world trajectories for training and multi-turn evaluation remains insufficiently understood.

In this work, **we study how to leverage real-world human–agent coding trajectories to improve coding-agent user experience through training and multi-turn evaluation.** This requires addressing two key challenges. First, trajectories vary widely in interaction and execution quality, making it difficult to select reliable training supervision without discarding informative failures, corrections, and recovery behavior. Second, trajectories are fixed records of past interactions rather than reusable executable tasks, limiting reproducible evaluation and controlled generation of new interactions. Recovering such tasks requires reconstructing and validating repository states, tests, and execution environments from incomplete trajectory evidence.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

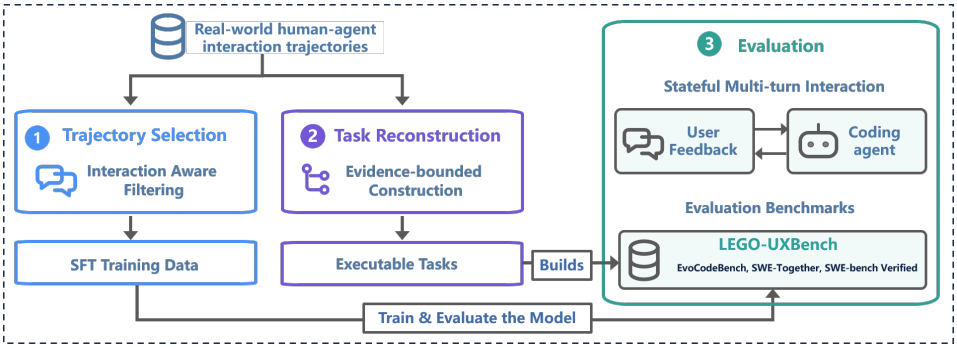


Figure 1: Overview of LEGO-UX, a three-component framework for learning from human–agent trajectories and evaluating coding-agent user experience. (1) **Trajectory selection** applies interaction-aware filtering to identify reliable trajectories for supervised fine-tuning. (2) **Task reconstruction** recovers and validates executable tasks from trajectory, which are then used to construct LEGO-UXBENCH. (3) **Evaluation** simulates stateful multi-turn interaction with user feedback and evaluates user experience by jointly considering test-verified task completion and interaction quality.

We introduce LEGO-UX, a framework that combines interaction-aware trajectory selection, evidence-bounded task reconstruction, and multi-turn evaluation to improve coding-agent user experience. Figure 1 provides an overview. For trajectory selection, we analyze behavioral and execution signals to characterize how users refine requirements and provide feedback, how agents respond and adapt, and which trajectories provide reliable supervision for multi-turn training. For task reconstruction, we recover task provenance, repository states, tests, and reference changes, and admit only candidates that satisfy executable validation. We further construct LEGO-UXBENCH, a 95-task benchmark from open-sourced trajectories that evaluates both test-verified task completion and interaction quality, with broader task coverage reflecting diverse real-world usage and complementing existing benchmarks.

We conduct comprehensive experiments to evaluate our framework and its key components. Across 11 frontier models on LEGO-UXBENCH, models with similar test-based task completion can differ substantially in interaction quality. Our trajectory analysis also shows that richer multi-turn interactions do not necessarily indicate more reliable completion, motivating careful selection of training supervision. Fine-tuning Qwen3.5-35B-A3B-Base (Qwen Team, 2026) on selected trajectories improves both measured dimensions and raises the unweighted eight-metric macro score across four benchmarks by 15.2 points. Controlled ablations validate the key design choices in our selection strategy. Our contributions are as below.

- We introduce LEGO-UX, a framework for improving coding-agent user experience through trajectory selection, task reconstruction, and multi-turn evaluation.
- We systematically analyze real-world human–agent trajectories and develop an interaction-aware selection method for identifying reliable training supervision.
- We reconstruct and validate executable tasks from real-world trajectories and construct LEGO-UXBENCH, a 95-task benchmark for multi-turn user-experience evaluation.
- We evaluate 11 frontier models using test-based and interaction-quality measures and fine-tune Qwen3.5-35B-A3B on selected trajectories, increasing the unweighted eight-metric macro score by 15.2 points.

2 PROBLEM FORMULATION AND MOTIVATION

Multi-turn interactive coding task. We study multi-turn interactive coding (Shen et al., 2026; Wu et al., 2026b), formalized as $\phi = (\theta, \pi_u, H)$, where $\theta = (x, \mathcal{R}, \mathcal{T}, \mathcal{E})$ is a repository-level coding task (Jimenez et al., 2024; Deng et al., 2025), with specification x, initial repository $\mathcal{R}$, test suite $\mathcal{T}$, and execution environment $\mathcal{E}$; π_u is the user policy; and H is the maximum number of interaction turns. Let s_t denote the workspace state after turn t, with s_0 initialized from $\mathcal{R}$. The binary verifier $P_\theta(s)$ equals 1 if all tests in $\mathcal{T}$ pass under $\mathcal{E}$, and 0 otherwise. At each turn t, the user policy generates a message conditioned on the task context, interaction history, and current s_{t-1}. The agent responds by modifying the repository, transitioning from s_{t-1} to s_t. Figure 5 illustrates this interaction process.

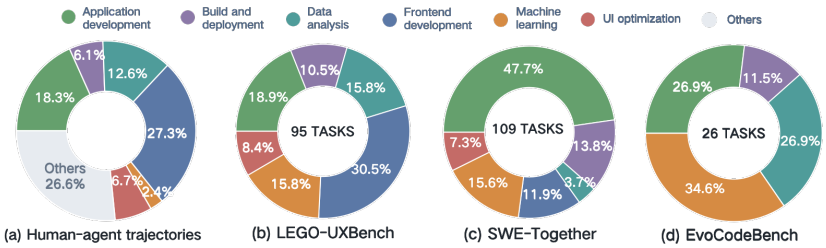


Figure 2: Task distributions across real-world human–agent trajectories and benchmarks. LEGO-UXBENCH spans a broader range of task types that better reflects real-world usage.

Motivation. Real-world human-agent trajectories provide a natural basis for studying and improving coding-agent user experience, capturing how users refine requests and how agents respond to feedback. In practice, effective assistance depends not only on task completion, but also on following evolving intent, responding to feedback, verifying progress, and reducing user effort (Baumann et al., 2026; Wu et al., 2026b). Recent work has separately studied real-world trajectories, interactive evaluation, and trajectory-to-environment reconstruction (Baumann et al., 2026; Shen et al., 2026; Wu et al., 2026a), yet current benchmarks still capture only part of the interaction and task diversity observed in real usage (see Figure 2), while raw trajectories vary in quality and are not directly reusable as executable tasks (Wu et al., 2026a). These gaps motivate a unified approach grounded in real-world human–agent trajectories: selecting reliable data for training, reconstructing their underlying tasks for benchmark construction, and simulating stateful multi-turn user–agent interactions for evaluation.

3 INTERACTION-AWARE TRAJECTORY ANALYSIS AND SELECTION

To systematically characterize real-world human–agent trajectories and identify reliable trajectories for training, we develop an interaction-aware data curation pipeline that integrates structural, behavioral, and execution signals, as illustrated in Figure 3.[1] The pipeline first annotates each trajectory with task, interaction, execution, and completion evidence, and then progressively constructs the training pool through three stages: *BasicDrop* enforces structural, safety, and privacy requirements; *HardDrop* applies deterministic filtering to the raw message–tool structure; and *FineHard* uses the annotations for evidence-based filtering and ranking with the Agent Productivity Score (APS).

3.1 TRAJECTORY TAGGING AND ANALYSIS

We characterize each trajectory from three complementary perspectives. *Execution evidence* captures observable agent behavior by normalizing interactions into events such as file access, code modifications, test and verification runs, execution outcomes, and terminal responses. *Task and interaction characteristics* use schema-constrained semantic annotation to identify the task type and area, and to assess action alignment and whether the interaction is explanation- or chat-only. *Completion dynamics* characterize task progress and completion by tracking required code modifications, verification outcomes, and unresolved failures. Details and statistics are provided in Appendix A.

3.2 TRAJECTORY SELECTION

BasicDrop: validity and safety filtering. BasicDrop enforces the minimum requirements for inclusion in the training corpus. It removes structurally invalid or unconvertible trajectories, including malformed records, invalid interactions, and corrupted content. It also performs an additional validation pass to re-screen for residual privacy and safety risks.

HardDrop: deterministic trajectory filtering. HardDrop applies deterministic criteria directly to the raw message–tool structure. It removes trajectories with insufficient interaction turns, incomplete or unsuccessful sessions, platform-access failures, invalid tool usage, or non-coding requests such as reasoning analysis, translation, title generation, and trajectory scoring. A trajectory is retained only if it passes all active criteria, with rejection reasons recorded for auditing.

[1]To protect privacy and mitigate security risks, all trajectories were anonymized before analysis, training, and evaluation, with all identifiable and sensitive information removed. Trajectories are used solely for research.

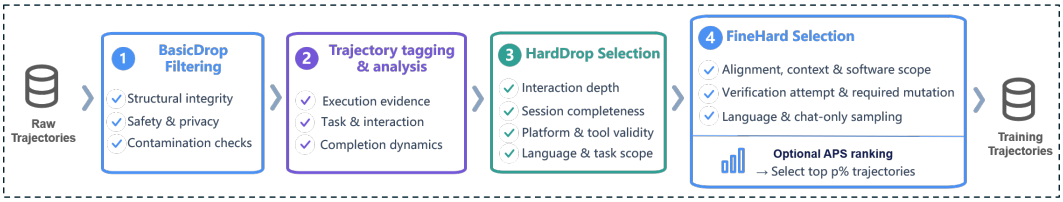


Figure 3: Interaction-aware data curation pipeline. Raw user–agent trajectories undergo BasicDrop filtering, trajectory tagging and analysis, and successive HardDrop and FineHard selection. FineHard applies APS ranking to retain the top p% trajectories, yielding the final training set.

FineHard: interaction and execution filtering FineHard further refines the trajectories using annotation-derived signals of interaction and execution quality. It requires agent actions to align with the user request, sufficient interaction context, and at least one verification attempt. When code changes are required, we additionally require evidence of a successful modification. Explanation- or chat-only trajectories are downsampled rather than excluded to preserve these common user interactions while prioritizing trajectories with substantive coding and execution evidence.

Note that these execution criteria do not require successful task completion: a verification attempt may fail, and a successful modification indicates only that a code change was applied. FineHard thus retains trajectories that contain failures, user corrections, or recovery behavior. We also consider stricter variants that exclude failed final verification or unresolved failures in our ablation studies.

APS: quality ranking with agent productivity score. FineHard applies binary criteria to interaction and execution evidence, but does not distinguish the strength of evidence among trajectories. We thus introduce the Agent Productivity Score (APS) to rank them along three complementary dimensions: verification quality (V) measures evidence that code changes behave as intended, execution cleanliness (X) captures execution failures and subsequent recovery, and change productivity (C) reflects the extent of successful code modification. We define APS as the weighted combination.

$$\text{APS} = w_V V + w_X X + w_C C, \qquad w_V, w_X, w_C \geq 0, \quad w_V + w_X + w_C = 1, \tag{1}$$

where $V, X, C \in [0, 1]$ and w_V, w_X, and w_C control their relative contributions. We use $(w_V, w_X, w_C) = (0.50, 0.35, 0.15)$ by default. We prioritize verification since it provides the most direct evidence that the resulting changes behave as intended, followed by execution cleanliness, which reflects failures and recovery without directly establishing correctness. Change productivity is weighted least because successful modifications indicate progress but provide weaker evidence of correctness. We evaluate the contribution of each component through ablations.

4 Evidence-Bounded Task Reconstruction

Human–agent trajectories provide valuable data for training and evaluation, but often lack sufficient information to recover reproducible repository states, tests, and execution environments. This limits reliable outcome verification, controlled trajectory generation, and reproducible evaluation. To address this limitation, we develop an evidence-bounded task reconstruction pipeline that recovers and validates executable tasks from available trajectory evidence, together with reference solutions representing the code changes that address the tasks. Figure 4 illustrates the three-stage pipeline.

Stage 1: Normalization and task anchoring. Given an interaction trajectory, the source adapter extracts and normalizes the user query, GitHub repository URL, commit identifier, and reference solution into a common representation. The repository URL and commit identifier anchor reconstruction to the corresponding commit, whose parent revision provides the pre-change repository state and whose changes provide the reference solution.This provenance-based anchoring supports subsequent task reconstruction and executable validation.

Stage 2: Reverse reconstruction. The pipeline then uses the normalized trajectory and repository anchor to locate the specified repository and commit, reconstruct the task instruction from the user query, and recover the reference solution from the commit. Test cases are obtained either from tests introduced by the commit or generated from the user query. The recovered repository state, instruction, tests, reference solution, and execution environment are assembled into a standardized, isolated task package for executable validation. Candidates without a recoverable reference solution are excluded from executable-task reconstruction.

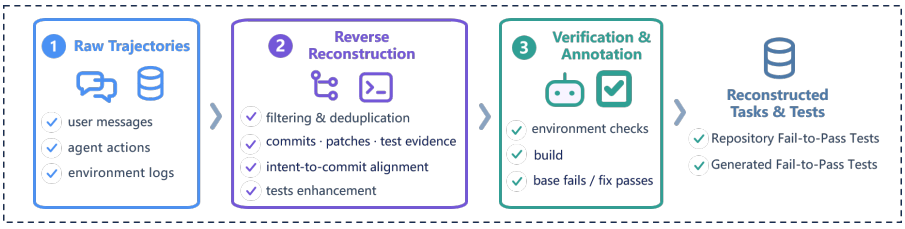


Figure 4: Evidence-bounded task reconstruction pipeline. It recovers task provenance, reconstructs the repository state and execution environment, and validates each candidate . Tests are recovered from the repository or generated when unavailable; unvalidated trajectories remain training-only.

Stage 3: Executable validation. The pipeline validates each reconstructed candidate by replaying it in the recovered base state s_0 and after applying the reference solution Δ. A candidate is admitted as an executable task only if the base state fails the verifier while the reference-patched state passes, i.e., $P_\theta(s_0) = 0$ and $P_\theta(s_0 \oplus \Delta) = 1$. Candidates satisfying this fail-to-pass criterion are assigned to the gold fail-to-pass tier (Jimenez et al., 2024), while candidates that lack a recoverable reference solution are not treated as executable benchmark tasks. Validated tasks are then materialized as isolated harbor cases containing the reconstructed instruction, environment, verifier, and provenance, with the reference solution hidden from the agent.

5 Executable Multi-Turn Evaluation Protocol



Figure 5: Overview of multi-turn human–agent interaction. Starting from an initial user query, the coding agent iteratively responds to user requests. At each turn, the agent modifies and verifies the workspace, while an LLM-simulated user generates subsequent feedback based on the execution.

We evaluate multi-turn coding-agent user experience by simulating stateful user–agent interactions, constructing a new benchmark, and measuring both task completion and interaction quality.

5.1 Stateful User–Agent Simulation

To evaluate coding agents under stateful multi-turn interactions, we simulate evolving user requests over a persistent workspace within the Harbor framework. Specifically, each task $\phi = (\theta, \pi_u, H)$ starts with specification x, and a workspace initialized from s_0. After each agent turn t, the resulting state s_t is evaluated and the user simulator π_u generates a follow-up request based on the interaction history and the current state. The process continues until the horizon H is reached or the stopping criterion is satisfied. The workspace and agent context persist across turns. In all reported evaluations, the user simulator is GPT-5.5 (OpenAI, 2026a), with its prompt and decoding configuration fixed across evaluated agents.

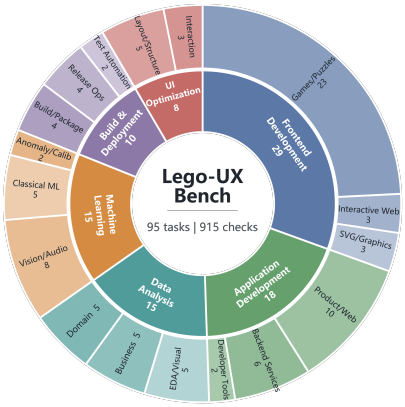
The user simulator generates follow-up requests with four interaction intents: (a) requesting debugging, (b) requesting stronger tests, (c) reporting a failure, and (d) clarifying a requirement. To generate context-aware feedback, the simulator observes a bounded interaction history, including user requests, agent responses and tool summaries, modified files, and patch excerpts. It may optionally receive a sanitized verification signal indicating whether the current solution passes the verifier. To prevent leakage, test names, assertions, file paths, raw test outputs, and hidden-test content are withheld.

5.2 Benchmark Construction

Existing coding benchmarks capture complementary aspects of agent performance. SWE-bench Verified evaluates single-turn resolution of real GitHub issues, SWE-Together reconstructs interactive tasks from real user–agent sessions, and EvoCode-Bench evaluates persistent implementation under authored requirement chains (Jimenez et al., 2024; Wu et al., 2026b; Shen et al., 2026). Figure 6

Benchmark	Interaction			Source	Evaluation			Tasks
	MT	PW	AF	TD	TE	IQ	EV	
SWE-bench Verified	×	×	×	×	×	×	✓	500
SWE-Together	✓	✓	✓	✓	×	✓	✓	109
EvoCode-Bench	✓	✓	×	×	✓	×	✓	26
LEGO-UXBENCH	✓	✓	✓	✓	✓	✓	✓	95

Figure 6: Comparison with representative coding-agent benchmarks. Columns summarize the interaction protocol, task provenance, and evaluation design; Appendix Table 13 provides full definitions and coding criteria.



compares their evaluation designs; detailed definitions of the comparison dimensions are provided in Appendix Table 13. Our analysis of real-world coding trajectories reveals substantial variation across task types, execution surfaces, and technical topics. Motivated by this gap, we construct LEGO-UXBENCH as a complementary benchmark with broader task coverage.

Using our reconstruction pipeline, we recover and validate executable tasks from CC-Bench Trajectories (Z.ai, 2025), GitHub PR Agent Trajectories (PotatoHD, 2026), and SWE-chat (SALT-NLP, 2026). LEGO-UXBENCH includes 95 tasks satisfying our executable evidence criteria, spanning six categories and four programming languages, with 915 deterministic test cases. The six categories—application development, build and deployment, data analysis, frontend development, machine learning, and UI optimization—cover diverse subcategories, as visualized in Figure 6. Detailed benchmark composition, comparison-dimension definitions, and task comparisons are in Appendix B

5.3 USER EXPERIENCE EVALUATION METRICS

We evaluate coding-agent user experience along two complementary dimensions: test-verified task completion and interaction quality. The former measures whether the task is successfully completed, while the latter assesses how effectively the agent follows and adapts to user feedback.

Test-based pass rate. Consider N evaluation tasks, where each task θ_i is evaluated through one or more independent rollouts. For task i and turn t, let $s_{i,t}$ denote the resulting state and $P_{i,t} := P_{\theta_i}(s_{i,t})$ denote its verifier results. Let $P_i^{(1)}$ and $P_i^{(2)}$ denote the final verifier results for task i. We define:

$$\mathrm{Pass}_t = \frac{1}{N}\sum_{i=1}^{N} P_{i,t}, \qquad \mathrm{Pass}^2 = \frac{1}{N}\sum_{i=1}^{N} P_i^{(1)} P_i^{(2)}.$$

Here, Pass_t measures the fraction of tasks fully resolved after turn t, and Pass^2 measures the fraction of tasks resolved in both independent rollouts, with higher values indicating more consistent performance across attempts. To characterize recovery over the interaction, let F denote the final interaction turn. We define $\mathrm{Recovery} = \frac{\sum_{i=1}^{N}(1-P_{i,1})P_{i,F}}{\sum_{i=1}^{N}(1-P_{i,1})}$ as the fraction of initially unresolved tasks resolved by the final turn.

LLM-based interaction quality. Test-based metrics capture the task-solving capability of coding agents but do not fully reflect the quality of human–agent interactions. We thus use an LLM judge to assess interaction quality as a proxy for human preferences (Zheng et al., 2023; Dou et al., 2025; Seshadri et al., 2026). In all reported evaluations, the interaction-quality judge is GPT-5.5 (OpenAI, 2026a), using a fixed schema-constrained prompt across models. The judge assigns each interaction an overall quality score $\mathrm{IQ}_i \in [0, 1]$ and six diagnostic scores covering task success, requirement following, context continuity, correction handling, testing and verification, and user effort. We aggregate the overall scores using MeanIQ:

$$\mathrm{MeanIQ} = \frac{1}{N}\sum_{i=1}^{N} IQ_i.$$

We also report an Elo-style metric for relative performance based on pairwise judgments. For each task, the judge compares two models' trajectories, and a global Bradley–Terry model aggregates

Table 1: Performance on LEGO-UXBENCH. Boldface indicates the best result in each column.

Model	Test-based Pass Rate				LLM-based Interaction Quality	
	Pass_1	Pass_F	Pass^2	Recovery	MeanIQ	Elo
GPT-5.6 Sol	**68.4%**	**68.4%**	54.7%	5%	**0.755**	1024.5
Claude Opus 4.8	65.8%	**68.4%**	60.0%	9%	0.713	1099.6
Claude Opus 4.7	64.7%	67.4%	**61.1%**	**11%**	0.714	1035.3
MiMo-V2.6-Pro	63.2%	65.8%	60.0%	**11%**	0.705	1099.5
GPT-5.5	64.2%	65.8%	**61.1%**	6%	0.748	1012.9
Kimi K3	61.6%	64.7%	55.8%	8%	0.722	**1105.2**
GLM-5.2	60.5%	62.1%	51.6%	8%	0.681	1020.2
DeepSeek V4 Pro	59.0%	60.5%	56.8%	5%	0.735	987.2
Kimi K2.7	55.3%	57.9%	50.5%	5%	0.682	967.6
DeepSeek V4 Flash	52.6%	57.4%	45.3%	9%	0.645	1031.3
DeepSeek V3.2	33.7%	39.5%	29.5%	5%	0.435	773.6

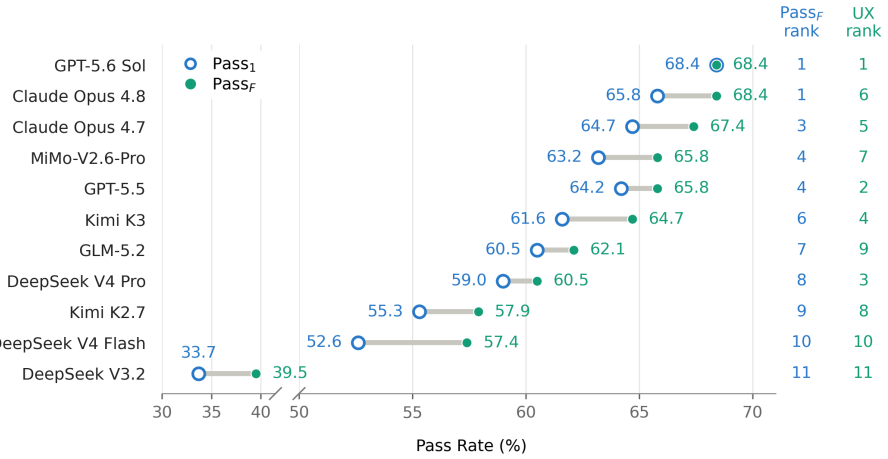


Figure 7: Pass rate and interaction quality on LEGO-UXBENCH. Hollow/filled markers indicate Pass_1/Pass_F, with lines showing interaction gains. Right columns show model rankings.

the outcomes into ratings (Chiang et al., 2024). Higher ratings indicate stronger interaction quality relative to the evaluated model pool.

6 EXPERIMENTAL RESULTS

6.1 EXPERIMENTAL SETUP

Evaluated models. We evaluate Claude Opus 4.8 and Claude Opus 4.7 (Anthropic, 2026b); Kimi K3 and Kimi K2.7 (Kimi Team, 2026; Moonshot AI, 2026); GPT-5.6 Sol and GPT-5.5 (OpenAI, 2026b;a); DeepSeek-V4 Flash, DeepSeek-V4 Pro, and DeepSeek-V3.2 (DeepSeek, 2026; 2025); GLM-5.2 (Z.ai, 2026); MiMo-V2.6-Pro (Xiaomi, 2026); Qwen3.5-35B-A3B (Qwen Team, 2026).

Benchmarks and evaluation protocol. We evaluate on LEGO-UXBENCH, SWE-Together (Wu et al., 2026b), EvoCode-Bench (Shen et al., 2026), and SWE-bench Verified (Jimenez et al., 2024), using OpenCode 1.14.22 (Anomaly Innovations, Inc., 2026) and Claude Code 2.1.118 (Anthropic, 2026a) as coding-agent harnesses. Each benchmark is evaluated with two independent runs. We report Pass_1, Pass_F, Pass^2, Recovery, MeanIQ, and Elo as defined in Section 5.3, and CaseScore for UniPat EvoCodeBench. For the training experiments, we compare the base Qwen3.5-35B-A3B (Qwen Team, 2026) with its SFT variant trained on our curated interaction data. Before training, candidate samples are deduplicated against every evaluation benchmark at the repository, commit, patch, and semantic-nearest-neighbor levels; all detected overlaps are excluded. For ablations, we report the macro average over the displayed metrics.

6.2 FUNCTIONAL AND INTERACTION PERFORMANCE ON LEGO-UXBENCH

Table 1 reports functional and interaction performance on LEGO-UXBENCH, while Figure 7 compares first-turn and final success alongside the functional and interaction-quality rankings. Overall, interaction constantly improves functional success for most models: 10 of 11 models achieve a higher

Table 2: Ablation of successive trajectory-selection stages. Macro Avg. averages all reported metrics.

Model	LEGO-UXBENCH				SWE-Together		UniPat EvoCodeBench	SWE-Bench Verified	Macro
	MeanIQ	Pass^2	Pass_1	Pass_F	Pass_1	Pass^2	CaseScore	Pass_1	Avg.
Base	0.18	5.3%	12.8%	16.6%	0.0%	0.0%	7.3%	2.0%	7.75
Harddrop	0.32	12.6%	18.4%	25.5%	4.6%	4.6%	13.5%	39.0%	18.78 (+11.0)
+ FineHard	0.38	13.7%	21.2%	30.1%	9.1%	9.1%	9.1%	44.0%	21.79 (+14.0)
+ APS_{70}	0.37	11.6%	21.8%	28.1%	13.6%	4.6%	8.7%	42.0%	20.93 (+13.2)
+ APS_{50}	0.34	13.7%	24.0%	28.6%	15.9%	9.1%	11.0%	47.0%	22.91 (+15.2)

Table 3: Ablation of APS score components.

Model	LEGO-UXBENCH				SWE-Together		UniPat EvoCodeBench	SWE-Bench Verified	Macro
	MeanIQ	Pass^2	Pass_1	Pass_F	Pass_1	Pass^2	CaseScore	Pass_1	Avg.
APS	0.34	13.7%	24.0%	28.6%	15.9%	9.1%	11.0%	47.0%	22.91
APS/V	0.35	11.6%	19.4%	26.4%	15.9%	8.9%	9.5%	43.0%	21.21 (-1.70)
APS/X	0.31	16.8%	24.4%	28.2%	9.1%	4.6%	11.3%	45.0%	21.30 (-1.61)
APS/C	0.33	20.0%	22.4%	31.3%	6.8%	4.6%	12.3%	42.0%	21.55 (-1.36)

Pass_F than Pass_1, with an average gain of 2.6 percentage points. The largest gains are observed for DeepSeek V3.2 (+5.8) and DeepSeek V4 Flash (+4.8), whereas GPT-5.6 Sol remains unchanged at 68.4%. Final success, however, does not necessarily imply consistent success across turns: Pass^2 is on average 8.3 points below Pass_F, with GPT-5.6 Sol achieving 68.4% Pass_F but only 54.7% Pass^2, compared with 61.1% for Claude Opus 4.7 and GPT-5.5.

Functional success also does not fully capture interaction quality. GPT-5.6 Sol and Claude Opus 4.8 both achieve 68.4% Pass_F, yet their MeanIQ differ substantially (0.76 vs. 0.71), corresponding to interaction-quality ranks 1 and 6. Similarly, GPT-5.5 and MiMo-V2.6-Pro both achieve 65.8% Pass_F, but rank 2 and 7 in interaction quality with MeanIQ of 0.75 and 0.71, respectively. Conversely, DeepSeek-V4-Pro ranks 8th in Pass_F (60.5%) but 3rd in interaction quality (0.74). These differences demonstrate that functional success and interaction quality capture complementary aspects of interactive coding performance, motivating their joint evaluation in LEGO-UXBENCH.

6.3 ABLATION ANALYSIS OF TRAJECTORY SELECTION PIPELINE

We conduct controlled ablations to assess the key components of our data-curation pipeline. We report the macro average across displayed metrics and its change relative to the corresponding baseline.

Ablation on selection stages. Table 2 ablates the successive stages of our trajectory-selection pipeline. HardDrop provides the initial deterministic filtering, FineHard introduces evidence-based filtering, and APS further ranks the retained trajectories by the Agent Productivity Score, with APS_{70} and APS_{50} retaining the top 70% and 50%, respectively. HardDrop substantially improves over the base model, increasing the macro-average score by 11.0 percentage points. Adding FineHard increases this gain to 14.0 points, with improvements across most metrics. APS_{50} achieves the largest overall gain of 15.2 points, including the highest LEGO-UXBENCH Pass_1, SWE-Together Pass_1, EvoCodeBench CaseScore, and SWE-bench Verified Pass_1. In contrast, APS_{70} yields a smaller macro-average gain of 13.2 points, indicating that the effect of APS selection depends on the retention ratio. Overall, these results demonstrate the complementary benefits of successive selection stages and the importance of controlling selection strength.

Ablation on APS ranking. Table 3 evaluates the contribution of the three APS components by removing verification quality V, execution cleanliness X, or change productivity C. Full APS achieves the highest macro average of 22.91. Removing any component reduces aggregate performance, with the largest decrease from removing V by 1.70 points, followed by X by 1.61 points and C by 1.36 points. These results show that all three components contribute to APS-based selection, with verification quality providing the largest aggregate contribution in this setting.

Ablation on supervision reliability. Table 4 compares two reliability-oriented variants of FineHard. *Exclude Verify-fail* removes trajectories whose final observed verification fails. *Failure-resolved* keeps data without unresolved failures. Excluding verification failures reduces the macro average from 21.8 to 21.0, despite gains on several individual metrics. *Failure-resolved* achieves the highest macro

Table 4: Ablation of reliability-oriented FineHard variants.

Selection recipe	LEGO-UXBENCH				SWE-Together		UniPat EvoCodeBench	SWE-Bench Verified	Macro
	MeanIQ	Pass^2	Pass_1	Pass_F	Pass_1	Pass^2	CaseScore	Pass_1	Avg.
FineHard	0.38	13.7%	21.2%	30.1%	9.1%	9.1%	9.1%	44.0%	21.8
+ exclude Verify-fail	0.37	14.7%	23.4%	29.0%	4.6%	0.0%	9.5%	50.0%	21.0 (-0.8)
+ failure-resolved	0.32	20.0%	24.6%	31.5%	9.1%	4.6%	11.7%	43.0%	22.1 (+0.3)

average of 22.1 with 0.3 points improvement, but with benchmark-dependent trade-offs. Overall, these results suggest that reliability-aware selection can modestly improve aggregate performance, while categorically excluding verification-failure trajectories is less effective in this setting.

7 RELATED WORK

Executable and interactive coding tasks. Coding-agent evaluation has progressed from executable repository tasks toward multi-turn interactions. SWE-bench and SWE-Bench Pro evaluate software-engineering tasks with executable verification (Jimenez et al., 2024; Deng et al., 2025), while SWE-Gym, R2E-Gym, SWE-smith, SWE-rebench, and CLI-Universe develop scalable approaches for constructing tasks (Pan et al., 2025; Jain et al., 2025; Yang et al., 2025; Badertdinov et al., 2025; Hua et al., 2026). Complementarily, SWE-chat captures real human–agent coding sessions (Baumann et al., 2026), while EvoCode-Bench, SWE-Interact, and SWE-Together evaluate agents under evolving user feedback (Shen et al., 2026; Raghavendra et al., 2026; Wu et al., 2026b). Concurrent work Terminal-Universe converts agent trajectories into reusable terminal tasks and multi-turn supervision (Wu et al., 2026a). However, reconstructing these multi-turn tasks directly from real-world interactions remains underexplored. We thus study the reconstruction of multi-turn coding tasks directly from observed human–agent interactions, enabling reproducible evaluation while preserving the interaction context.

User-centered and interaction-aware evaluation. Executable tests assess whether an implementation satisfies its specification, but do not capture the quality of interactions. ArtifactsBench makes a related distinction for visual code generation by combining executable evaluation with multimodal judging and human calibration (Zhang et al., 2025), while REAgent highlights the importance of understanding user requirements in software-engineering tasks (Kuang et al., 2026). We instead focus on stateful human–agent interaction alongside repository-level correctness. Since simulated users may not faithfully reflect human behavior (Dou et al., 2025; Seshadri et al., 2026) and LLM judges are not correctness oracles (Zheng et al., 2023), we evaluate these dimensions separately: executable tests measure task success, while interaction-based evaluation assesses communication, continuity, and responsiveness to user feedback.

Trajectory selection and supervision. Agent trajectories provide supervision beyond final patches by capturing intermediate reasoning, tool use, failures, and recovery. Prior work has explored supervised fine-tuning on validated coding trajectories (Tao et al., 2026), harness-native reinforcement learning (Du et al., 2026), and broader post-training strategies (Qiu et al., 2025; Wang et al., 2026; Xuejun et al., 2026). However, real human–agent trajectories introduce an additional challenge: interaction-rich behaviors such as follow-up, correction, and recovery do not necessarily imply reliable supervision. Moreover, executable evidence can itself be imperfect; for example, generated tests may lack sufficient discriminative power (Sun et al., 2026). We thus treat interaction value and supervision reliability as distinct considerations, and select trajectories using behavioral, execution, and verification signals while retaining informative failures, corrections, and recovery behavior.

8 CONCLUSION

We study how real-world human–agent coding trajectories can support training and multi-turn evaluation of coding-agent user experience. We introduce LEGO-UX, which combines interaction-aware trajectory selection, evidence-bounded task reconstruction, and stateful evaluation, and construct LEGO-UXBENCH from validated trajectory-derived tasks. Our results show that interaction-rich trajectories do not necessarily provide reliable supervision, while functional success and interaction quality capture complementary aspects of agent performance. Fine-tuning on selected trajectories further improves performance across benchmarks.

AI USE STATEMENT

Generative AI tools were used solely for language editing and stylistic refinement of author-written text. The authors conceived the research ideas, designed and conducted all experiments, analyzed the results, and wrote the manuscript. All AI-assisted language edits were reviewed and approved by the authors, who take full responsibility for the final manuscript.

ETHICS STATEMENT

This work uses previously collected human–agent coding trajectories and repository artifacts; it does not recruit participants or collect new human-subject data. Benchmark construction uses only open-source trajectories and repositories, subject to their access conditions and licenses. Trajectory analysis, selection, training, and ablations use a separate collection of internal real-world interactions. Access to these internal trajectories is restricted to the research workflow, and raw trajectories are not released; the paper reports only aggregate statistics and model-level results.

Coding trajectories can contain personal information, credentials, proprietary paths, or copyrighted code. Before analysis, training, and evaluation, we remove identified personal and sensitive information and filter exposed credentials and identity-contaminated or malformed records. We do not attempt to infer user identities. To reduce evaluation contamination, candidate training samples are compared against every benchmark at the repository, commit, patch, and semantic-nearest-neighbor levels, and all detected overlaps are excluded before training. These safeguards reduce, but cannot eliminate, residual privacy, licensing, or contamination risks; released artifacts therefore exclude internal trajectories and internal filesystem metadata.

Both the user simulator and interaction-quality judge use GPT-5.5. Their outputs may reflect model- and prompt-specific biases, and using the same model family for both roles may introduce correlated measurement error. We therefore treat interaction-quality scores as model-based proxies rather than direct measurements of human preference, report them separately from executable verification, and describe this limitation explicitly. The benchmark is intended for research in isolated execution environments; coding-agent outputs should be reviewed before deployment, especially when they can modify repositories, execute commands, or access external systems.

REPRODUCIBILITY STATEMENT

Implementation and experimental details are provided in Sections 4–6 and the appendix. We will release the code, configurations, benchmark tasks, and evaluation scripts needed to reproduce our main results. Repository revisions and execution environments are versioned where applicable to support reproducible evaluation.

REFERENCES

Anomaly Innovations, Inc. OpenCode, 2026. URL https://github.com/anomalyco/opencode/releases/tag/v1.14.22. Version 1.14.22.

Anthropic. Claude Code, 2026a. URL https://github.com/anthropics/claude-code/releases/tag/v2.1.118. Version 2.1.118.

Anthropic. Introducing Claude Opus 4.7, April 2026b. URL https://www.anthropic.com/news/claude-opus-4-7.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents, 2025. URL https://arxiv.org/abs/2505.20411.

Joachim Baumann, Vishakh Padmakumar, Xiang Li, John Yang, Diyi Yang, and Sanmi Koyejo. SWE-chat: Coding agent interactions from real users in the wild, 2026. URL https://arxiv.org/abs/2604.20779.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael I. Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating LLMs by human preference. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 8359–8388. PMLR, 2024.

DeepSeek. DeepSeek-V3.2 Release, December 2025. URL https://api-docs.deepseek.com/news/news251201/.

DeepSeek. DeepSeek-V4-Pro GA Release, August 2026. URL https://api-docs.deepseek.com/news/news260813/.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks?, 2025. URL https://arxiv.org/abs/2509.16941.

Yao Dou, Michel Galley, Baolin Peng, Chris Kedzie, Weixin Cai, Alan Ritter, Chris Quirk, Wei Xu, and Jianfeng Gao. SimulatorArena: Are user simulators reliable proxies for multi-turn evaluation of AI assistants?, 2025. URL https://arxiv.org/abs/2510.05444.

Yiming Du, Yuxin Jiang, Tao Yuan, Jianbo Dai, Shaowei Wang, Jierun Chen, Chaofan Tao, Xianzhi Yu, Lifeng Shang, Kam-Fai Wong, Xiaohui Li, and Haoli Bai. LEGO-RL: Harness-native reinforcement learning for coding agents, 2026. URL https://arxiv.org/abs/2608.17393.

Zhanbo Hua, Yifan Yao, Weihao Xie, Yongchi Zhao, Minghao Liu, Ruizhi Qiu, Zhewei Huang, Zun Wang, Yiyan Ji, Yunhai Ye, Letian Zhu, Xinping Lei, Han Li, Zhiyuan Ma, Zili Wang, Zhaoxiang Zhang, and Jiaheng Liu. CLI-Universe: Towards verifiable task synthesis engine for terminal agents, 2026. URL https://arxiv.org/abs/2606.22883.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents, 2025. URL https://arxiv.org/abs/2504.07164.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Kimi Team. Kimi K3: Open frontier intelligence, 2026. URL https://arxiv.org/abs/2607.24653.

Shiqi Kuang, Zhao Tian, Kaiwei Lin, Chaofan Tao, Shaowei Wang, Haoli Bai, Lifeng Shang, and Junjie Chen. Reagent: Requirement-driven llm agents for software issue resolution. *arXiv preprint arXiv:2604.06861*, 2026.

Moonshot AI. Kimi K2.7 Code, 2026. URL https://www.kimi.com/resources/kimi-k2-7-code.

OpenAI. Introducing GPT-5.5, April 2026a. URL https://openai.com/index/introducing-gpt-5-5/.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition, July 2026b. URL https://openai.com/index/gpt-5-6/.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym, 2025. URL https://arxiv.org/abs/2412.21139.

PotatoHD. GitHub PR Agent Trajectories. Hugging Face dataset, 2026. URL https://huggingface.co/datasets/PotatoHD/github-pr-agent-trajectories.

Suming Qiu, Jing Li, Zhicheng Zhou, Junjie Huang, Linyuan Qiu, and Zhijie Sun. Logits replay + MoClip: Stabilized, low-cost post-training with minimal forgetting, 2025. URL https://arxiv.org/abs/2510.09152.

Qwen Team. Qwen3.5-35B-A3B. Model card, 2026. URL https://huggingface.co/Qwen/Qwen3.5-35B-A3B.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. SWE-INTERACT: Reimagining SWE benchmarks as user-driven long-horizon coding sessions, 2026. URL https://arxiv.org/abs/2606.30573.

SALT-NLP. SWE-chat. Hugging Face dataset, 2026. URL https://huggingface.co/datasets/SALT-NLP/SWE-chat.

Preethi Seshadri, Samuel Cahyawijaya, Ayomide Odumakinde, Sameer Singh, and Seraphina Goldfarb-Tarrant. Lost in simulation: LLM-simulated users are unreliable proxies for human users in agentic evaluations, 2026. URL https://arxiv.org/abs/2601.17087.

Haiyang Shen, Xuanzhong Chen, Wendong Xu, Yun Ma, Liang Chen, and Kuan Li. EvoCode-Bench: Evaluating coding agents in multi-turn iterative interactions, 2026. URL https://arxiv.org/abs/2605.24110.

Yuxuan Sun, Yuze Zhao, Yufeng Wang, Yao Du, Zhiyuan Ma, Jinbo Wang, Mengdi Zhang, Kai Zhang, and Zhenya Huang. SWE-Mutation: Can LLMs generate reliable test suites in software engineering?, 2026. URL https://arxiv.org/abs/2605.22175.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving, 2026. URL https://arxiv.org/abs/2601.01426.

Ruoyu Wang, Jierun Chen, Shaowei Wang, Chaofan Tao, Sidi Yang, Yuxin Jiang, Kim-Hui Yap, Lifeng Shang, Xiaohui Li, and Haoli Bai. Swe-review: Closing the loop on issue resolution with agentic code review. *arXiv preprint arXiv:2607.06065*, 2026.

Jie Wu, Zhenru Zhang, Beichen Zhang, Xuwu Wang, Yuhui Su, Mouxiang Chen, Peng Wang, Zhihai Wang, Que Shen, Hao Zhou, An Yang, Fei Huang, Yujiu Yang, and Dayiheng Liu. Terminal-Universe: Turning agent trajectories into scalable terminal environments, 2026a. URL https://arxiv.org/abs/2609.04148.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions, 2026b. URL https://arxiv.org/abs/2606.29957.

Xiaomi. Introducing MiMo-V2.6 series. https://mimo.xiaomi.com/mimo-v2-6, 2026.

Yu Xuejun, Jianyuan Zhong, Zijin Feng, Pengyi Zhai, Roozbeh Yousefzadeh, Wei Chong Ng, Haoxiong Liu, Ziyi Shou, Jing Xiong, Yudong Zhou, Claudia Beth Ong, Austen Jeremy Sugiarto, Yaoxi Zhang, Wai Ming Tai, Huan Cao, Dongcai Lu, Jiacheng Sun, Qiang Xu, SHEN XIN, and Zhenguo Li. Mathesis: Towards formal theorem proving from natural languages. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=CJdX82odge.

John Yang, Kilian Leret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents, 2025. URL https://arxiv.org/abs/2504.21798.

Z.ai. CC-Bench Trajectories. Hugging Face dataset, 2025. URL https://huggingface.co/datasets/zai-org/CC-Bench-trajectories.

Z.ai. GLM-5.2, 2026. URL https://huggingface.co/zai-org/GLM-5.2.

Chenchen Zhang, Yuhang Li, Can Xu, Jiaheng Liu, Ao Liu, Changzhi Zhou, Ken Deng, Dengpeng Wu, Guanhua Huang, Kejiao Li, Qi Yi, Ruibin Xiong, Shihui Hu, Yue Zhang, Yuhao Jiang, Zenan Xu, Yuanxing Zhang, Wiggin Zhou, Chayse Zhou, and Fengzong Lian. ArtifactsBench: Bridging the visual-interactive gap in LLM code generation evaluation, 2025. URL https://arxiv.org/abs/2507.04952.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-judge with MT-Bench and chatbot arena, 2023. URL https://arxiv.org/abs/2306.05685.

A TRAJECTORY DATA STATISTICS AND ANALYSIS

This section characterizes the tagged source-trajectory corpus used in Section 3 and establishes the empirical basis for our interaction-aware data selection strategy. We first summarize the principal findings on multi-turn interactions, followed by analyses of corpus composition, interaction structure, execution, recovery, and completion as conversation depth increases.

A.1 TRAJECTORY-LEVEL LABEL DISTRIBUTIONS

A schema-constrained language model assigns each trajectory four controlled attributes and two binary intent labels, summarized in Table 5. Task type, task area, and task topic capture distinct aspects of the underlying task and are assigned independently: task type denotes the requested operation, task area denotes the environment or domain in which the work is performed or delivered, and task topic denotes the primary technical subject. The language attribute identifies a single programming language, multiple programming languages, or no identifiable language. The two binary labels indicate whether the agent's actions are aligned with the user request and whether the trajectory is explanation- or chat-only.

Table 5: Trajectory-level label schema.

Label	**Meaning**	**Role**
Task type	Requested operation	Coverage
Task area	Execution or delivery surface	Coverage
Task topic	Primary technical subject	Coverage
Language	Identifiable programming language	Coverage
Action alignment	Actions pursue the request	Behavioral
Explanation/chat only	Non-action-bearing interaction	Behavioral

These attributes serve distinct roles in data curation. Task type, area, topic, and language characterize the coverage and composition of the corpus, while action alignment and explanation-only intent enable behavioral stratification for different training objectives. For example, an explanation-only trajectory may provide useful supervision for explanation or review behaviors but may be less-effective for executable-code SFT. The coverage attributes are descriptive and do not, by themselves, constitute quality judgments.

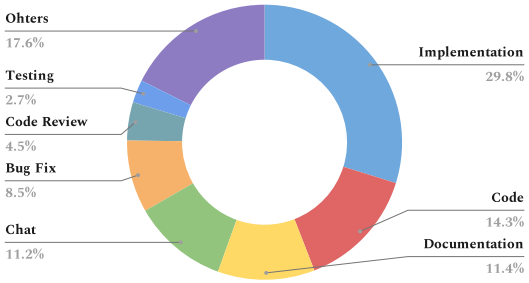


Figure 8: Distribution of task types. Implementation, code explanation, and documentation are the three most frequent task types.

Table 6: Task-type proportions in the tagged corpus.

Task type	Proportion (%)
implementation	29.8
code-explanation	14.3
documentation	11.4
meta-chat	11.2
bug-fix	8.5
non-software	8.4
code-review	4.5
build-or-ops	4.3
testing	2.7
refactor	2.7
data-or-ml	2.3

Task type. Task type captures what the user asks the agent to do. Each trajectory is assigned one label from a closed vocabulary of operations, such as `implementation`, `bug-fix`, `testing`, `code-review`, `documentation`, and `meta-chat`. The label describes the requested operation independently of where the work is performed or what technical subject it concerns. For example, implementing a bubble-sort CLI program and implementing a web page are both labeled `implementation`. As shown in Figure 8 and Table 6, `implementation` is the most common task type (29.8%), followed by code explanation, documentation, and meta-chat.

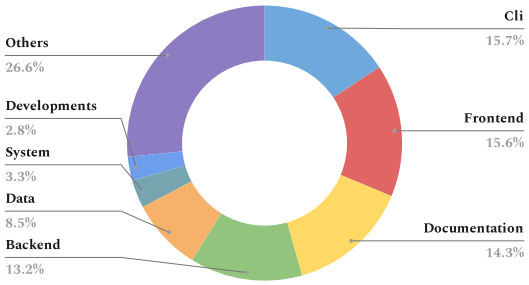


Figure 9: Distribution of task areas. CLI, frontend, documentation, and backend are the largest named execution surfaces.

Table 7: Task-area proportions in the tagged corpus.

Task area	Proportion (%)
non-code	17.6
cli	15.7
frontend	15.6
documentation	14.3
backend	13.2
data	8.5
library	4.8
systems	3.3
devops	2.8
machine-learning	2.4
fullstack	1.5
framework	0.3

Task area. Task area captures where the work is performed or delivered, namely, the execution surface, independently of the requested operation and technical subject. Each trajectory is assigned one label from a closed vocabulary, including `frontend`, `backend`, `cli`, `data`, `devops`, `documentation`, and `systems`. For example, a React testing task is labeled `frontend` because it targets a frontend application, while a bubble-sort program delivered as a command-line tool is labeled `cli`, regardless of the requested operation. As shown in Figure 9 and Table 7, non-code, `cli`, `frontend`, documentation, and `backend` are the most prevalent task areas.

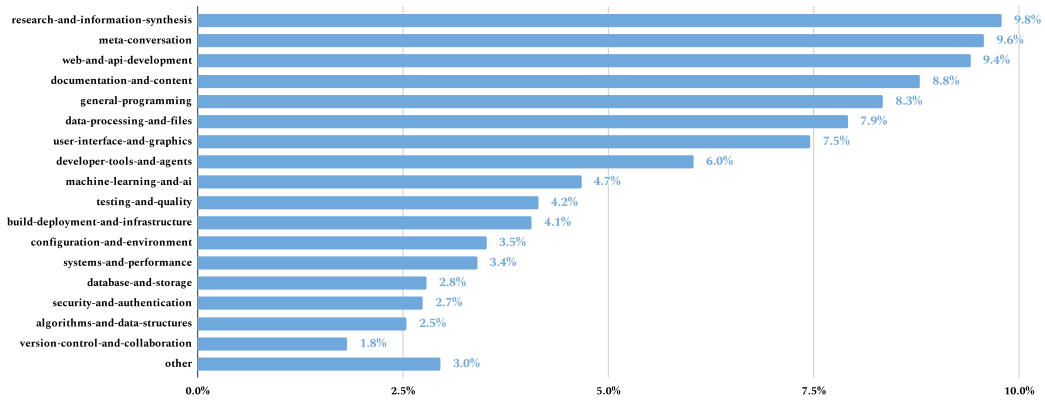


Figure 10: Distribution of task topics. Research and information synthesis, meta-conversation, and web and API development are the three most frequent task topics.

Task topic. Task topic captures the primary technical subject of the work, independently of the requested operation and execution surface. Each trajectory is assigned one label from a closed vocabulary of technical subjects, such as `research-and-information-synthesis`, `meta-conversation`, `web-and-api-development`, `general-programming`, and `database-and-storage`. For example, reviewing database pagination is labeled `database-and-storage` even though the task type is code review and the task area is backend; a React testing task is labeled `web-and-api-development` even though the task type is testing and the task area is frontend. As shown in Figure 10, the corpus spans a broad range of technical subjects, with research and information synthesis, meta-conversation, and web and API development being the three most frequent topics.

Programming language. The language attribute captures the primary identifiable programming language of the work, independently of task type, task area, and task topic. Each trajectory is assigned a single language when one language dominates, `multiple` for mixed-language work, or `none`

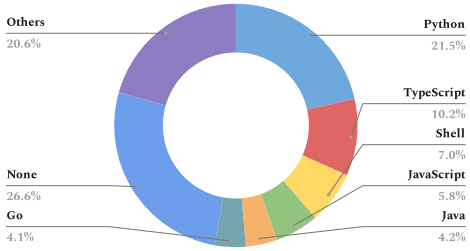


Figure 11: Distribution of programming languages. Python, TypeScript, and shell are the three most frequent named languages.

Table 8: Programming-language proportions in the tagged corpus.

Language	Proportion (%)
none	26.6
python	21.5
typescript	10.2
multiple	7.5
shell	7.0
javascript	5.8
java	4.2
go	4.1
c	3.3
cpp	2.7
unknown	2.2
html-css	2.1

when no programming language is identifiable. As shown in Figure 11, Python, TypeScript, and shell are the three most frequent named languages, accounting for 21.5%, 10.2%, and 7.0% of trajectories, respectively. Notably, 26.6% of trajectories have no identifiable language. The displayed categories cover 97.2% of the corpus, with the remaining 2.8% comprising low-frequency language categories.

Intent and actionability. We use two independent binary labels to distinguish behavioral alignment from task actionability. `agent-actions-aligned` indicates whether the agent's reasoning, actions, and response pursue the user request, irrespective of task completion; 92.9% of trajectories are aligned and 7.1% are not. `is-explanation-or-chat-only` separates explanation, review, summarization, and general chat from action-bearing requests such as creating, modifying, debugging, testing, or deploying software; 40.3% of trajectories are explanation- or chat-only, while 59.7% involve action-bearing work. These labels support corpus auditing and supervision-role assignment rather than serving as direct measures of trajectory quality or task completion.

A.2 Real-World Multi-Turn Interaction Dynamics

Building on the trajectory-level characteristics above, we next examine how real user–agent interactions evolve beyond the initial request and what these dynamics imply for data curation. We classify a trajectory as *single* when its normalized message statistics contain exactly one user message, and as *multiple* otherwise. We first characterize the prevalence and workload of multi-turn interactions, then examine how their turn semantics and requirements evolve, followed by analyses of execution, recovery, and completion.

A.2.1 Multi-Turn Structure and Turn Semantics

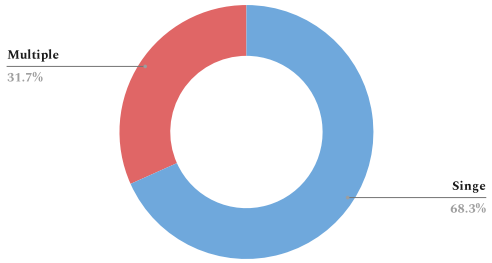


Figure 12: Single-turn versus multi-turn proportions in the tagged corpus.

Table 9: Average number of user messages, assistant turns, and tool calls per trajectory for single- and multi-turn cohorts in the tagged corpus.

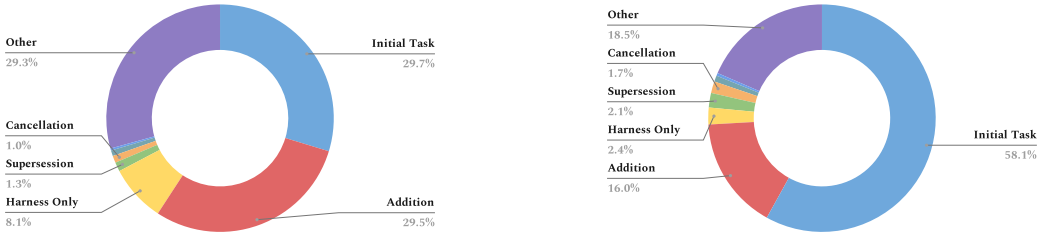
Cohort	User	Assistant	Tool
Single	1.00	10.45	11.44
Multiple	7.16	39.45	39.67

Turn prevalence and workload. Figure 12 and Table 9 summarize the prevalence and interaction workload of single- and multi-turn trajectories in the tagged corpus. The two cohorts account for 68.3% and 31.7% of trajectories, respectively. The table reports the mean number of user messages,

assistant turns, and tool calls per trajectory. Counts are aggregated over each complete trajectory before averaging across the cohort, rather than normalized by the number of user turns; consequently, the single-turn cohort has a mean of 1.00 user message by construction. Multi-turn trajectories involve substantially more assistant turns and tool calls, indicating considerably greater interaction and execution workload. Because these means can be influenced by a small number of very long trajectories, they describe overall workload rather than the behavior of a typical trajectory.

Table 10: User-turn-type proportions at the message and trajectory levels.

User-turn type	Message proportion (%)	Trajectory proportion (%)
Initial task	29.7%	87.2%
Addition	29.5%	24.1%
Harness only	8.1%	3.6%
Supersession	1.3%	3.1%
Cancellation	1.0%	2.5%
Meta	0.8%	1.3%
Answer	0.3%	0.6%
Confirmation	0.1%	0.3%
Feedback	0.1%	0.3%
Clarification	0.1%	0.2%
Non-task	29.0%	27.0%



(a) Message-level distribution

(b) Trajectory–type incidence distribution

Figure 13: User-turn-type distributions in the tagged corpus. The message-level distribution (left) partitions all user messages by type, while the trajectory–type incidence distribution (right) shows the relative frequency of trajectory–type incidences after normalization. Because a trajectory may contain multiple turn types, the right-hand distribution does not represent the proportion of trajectories containing each type.

User-turn composition. Table 10 and Figure 13 characterize the deterministic labels assigned to normalized user messages. Each message receives exactly one type. `initial-task` denotes the first actionable request; `addition` covers new requirements, additional actionable instructions, or requests to continue; `clarification` makes an existing requirement more precise; and `answer` provides information or a choice requested by the agent. `correction` revises an earlier instruction without changing its overall goal, while `supersession` replaces a prior requirement and `cancellation` withdraws one. `feedback` evaluates the agent's work, `confirmation` acknowledges a preceding interaction, and `meta` concerns the interaction itself. `harness-only` captures execution-system messages such as interruption notices, task notifications, and environment reminders. `non-task` covers messages without a confidently identifiable actionable task, such as greetings, empty messages, or ambiguous follow-ups.

The message proportion measures the fraction of all user messages assigned to each type, whereas the trajectory proportion measures the fraction of trajectories containing that type at least once. Message proportions therefore sum to approximately 100%, while trajectory proportions can overlap because a trajectory may contain multiple types. The message-level distribution is dominated by `initial-task` (29.7%), `addition` (29.5%), and `non-task` (29.0%). At the trajectory level, `initial-task` appears in 87.2% of trajectories, while `addition` and `non-task` appear in 24.1% and 27.0%, respectively. The right-hand visualization in Figure 13 normalizes the overlapping trajectory–type incidences for visualization and therefore should not be interpreted as trajectory proportions; the corresponding proportions are reported in Table 10.

A.2.2 Execution, Recovery, and Completion Outcomes

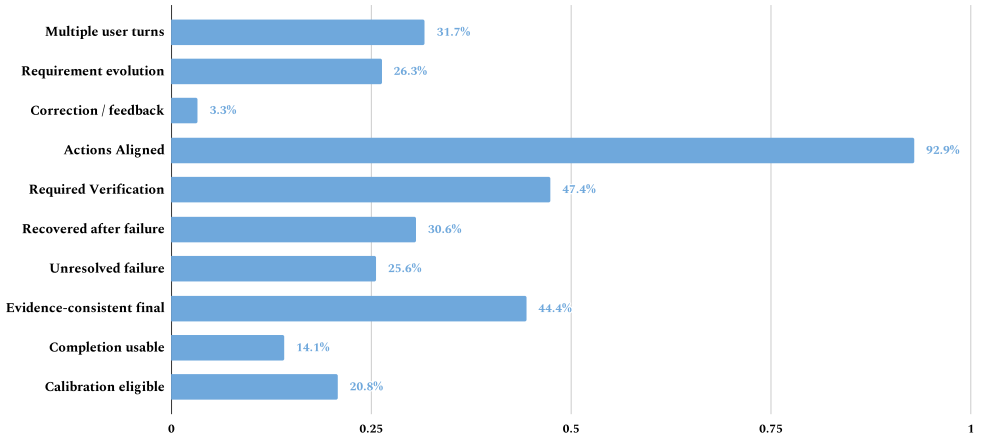


Figure 14: Trajectory-level interaction, execution, recovery, communication, and completion indicators in the combined corpus. Bars show the proportion of eligible trajectories satisfying each indicator. Eligibility is indicator-specific, so the reported rates use different denominators and are neither mutually exclusive nor additive.

Trajectory-level diagnostic indicators. Figure 14 summarizes ten trajectory-level diagnostic indicators covering user interaction, execution, verification, recovery, and completion. These indicators capture distinct properties of a trajectory and may co-occur; they are not mutually exclusive outcome labels. Because their definitions and eligible populations differ, we first describe each indicator before analyzing the observed patterns.

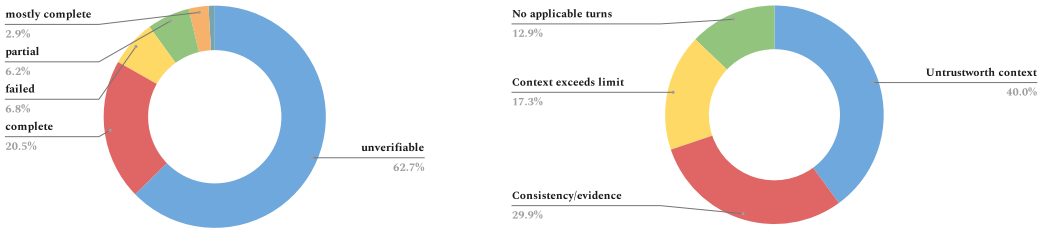
- **Multiple user turns**. A trajectory is counted if it contains more than one user message.
- **Requirement evolution**. A trajectory is counted if it contains an actionable or requirement-related follow-up turn, including an addition, clarification, answer, correction, supersession, cancellation, feedback, or confirmation.
- **Correction / feedback**. A trajectory is counted if it contains a correction, supersession, or feedback turn, capturing an explicit repair or override of a prior request.
- **Actions aligned**. A trajectory is counted if the agent's reasoning, actions, and final response are judged to pursue the user's request. It does not imply successful execution, or correctness.
- **Required verification attempted**. The eligible set consists of trajectories identified as requiring verification by the fused rule and LLM criteria. A trajectory is counted if it contains at least one observed verification run, such as a test, build, lint, type check, or runtime check.
- **Recovered after failure**. The eligible set consists of trajectories with at least one observed failure. A trajectory is counted if at least one failure is subsequently linked to a recovering action.
- **Unresolved failure**. A trajectory is counted if at least one observed failure has no subsequent recovering action. This indicator uses all completion-tagged trajectories as its denominator and is therefore not the complement of the recovery rate.
- **Evidence-consistent final answer**. A trajectory is counted if its final answer is judged consistent with the task, the actions taken, and the evidence visible in the trajectory.
- **Completion usable**. A trajectory is counted if the hybrid completion policy classifies it as either high-confidence complete or likely complete and does not flag it for review because of unverifiability, insufficient evidence, or disagreement between rule-based and LLM judgments.
- **Calibration eligible**. A trajectory is counted if its completion judgment is not excluded by the annotation context and has no session- or turn-level consistency hard blocker. This identifies trajectories whose LLM completion labels are eligible for claim-bearing use.

Multiple user turns occur in 31.7% of trajectories, with requirement evolution in 26.3% and explicit correction or feedback in 3.3%. Agent actions are judged aligned with user intent in 92.9% of eligible trajectories, whereas 44.4% of completion-tagged trajectories end with an evidence-consistent final answer and 25.6% retain an unresolved failure. Among trajectories requiring verification, 47.4% include a verification attempt; among those with an observed failure, 30.6% show recovery. Finally, 20.8% are calibration-eligible and 14.1% satisfy the completion-usable criterion. Overall, the gap between action alignment and completion-related indicators shows that alignment alone does not characterize completion usability. Because the indicators use different eligible populations, these marginal rates should not be interpreted as conditional relationships between indicators.

Table 11: Distribution of trajectory completion status and unverifiable outcomes. Completion-status proportions are computed over all completion-tagged trajectories. Reason proportions are conditional on trajectories labeled `unverifiable`. Percentages may not sum to exactly 100% because of rounding.

Status or reason	**Proportion**
Completion status *(all tagged trajectories)*	
`unverifiable`	**62.7%**
`complete`	20.5%
`failed`	6.8%
`partial`	6.2%
`mostly_complete`	2.9%
`not_attempted`	0.9%
Primary unverifiable reason *(unverifiable trajectories only)*	
`untrustworthy_task_context`	40.0%
`post-judge consistency / evidence blocker`	29.9%
`essential_context_exceeds_limit`	17.3%
`no_applicable_turns`	12.9%
`other deterministic abstention`	0.0%

Completion-status definitions. Table 11 and Figure 15 summarize the mutually exclusive completion-status labels assigned at the trajectory level. The completion judge prioritizes observable artifacts and tool results over unsupported statements in the final response and assigns completion obligations only when the user task is sufficiently trustworthy and applicable for judgment. A trajectory is labeled `complete` when the requested outcomes are judged satisfied without missing requirements or hard consistency blockers. `mostly-complete` and `partial` denote different degrees of incomplete fulfillment, without fixed numerical thresholds separating them. `failed` denotes an attempted but unsuccessful outcome, whereas `not-attempted` indicates the absence of a substantive attempt. Finally, `unverifiable` indicates that the available task context or evidence is insufficient to support a reliable completion judgment. Hard consistency blockers can further prevent a trajectory from receiving a `complete` label.



(a) Completion-status distribution

(b) Reasons conditional on `unverifiable`

Figure 15: Completion-status distribution and decomposition of unverifiable outcomes in the combined corpus. Panel (a) reports completion statuses over all completion-tagged trajectories, while panel (b) reports primary reasons conditional on trajectories labeled `unverifiable`.

Categorical completion outcomes. `Unverifiable` is the most frequent status, accounting for 62.7% of completion-tagged trajectories, while 20.5% are labeled `complete`. Explicitly unsuccessful or incomplete outcomes are less frequent: `failed`, `partial`, `mostly-complete`, and `not-attempted` account for 6.8%, 6.2%, 2.9%, and 0.9%, respectively. This distribution shows that the dominant obstacle to assigning a positive completion label is not an observed execution failure, but the inability to establish completion reliably from the available trajectory evidence.

The decomposition of `unverifiable` outcomes further clarifies this pattern. Untrustworthy task context (40.0%) and post-judgment consistency or evidence blockers (29.9%) together account for 69.9% of unverifiable trajectories, indicating that most abstentions arise from uncertainty about the task or insufficient support for the completion judgment. Another 17.3% result from essential context exceeding the judge's input limit, while 12.9% contain no applicable user turn that establishes a completion obligation. These results distinguish *failure to complete* from *failure to verify completion*: the former provides evidence of an unsuccessful agent outcome, whereas the latter indicates insufficient evidence to make a reliable determination. This distinction motivates treating `unverifiable` as a separate abstention category rather than merging it with `failed` or other non-complete outcomes.

A.2.3 Depth-Conditioned Behavior and Outcome Quality

To characterize behavior and outcome quality across conversation depths, we consider eight trajectory-level indicators. These indicators are derived using deterministic rules over the normalized trajectory (*rule*), schema-constrained LLM judgments (*LLM*), or hybrid criteria combining rule-based and LLM scores (*hybrid*). Unlike the mutually exclusive completion-status labels above, these indicators capture distinct properties of a trajectory and may co-occur. Figure 16 compares their rates across single- and multi-turn cohorts.

- **Aligned** (*LLM*). Whether the agent's reasoning, actions, and final response pursue the user's request, regardless of execution success.
- **Mutation** (*rule*). Whether the trajectory contains at least one successful file mutation, including an edit, write, or applied patch.
- **Completion coverage** (*structural*). Whether completion-stage annotations are available for the trajectory. The four indicators below are computed only for trajectories with these annotations.
  - **Requirement evolution** (*rule*). Whether the trajectory contains an actionable or requirement-related follow-up, including an addition, clarification, answer, correction, supersession, cancellation, feedback, or confirmation.
  - **Final consistent** (*LLM*). Whether the final answer is judged consistent with the task, the actions taken, and the visible evidence.
  - **Unresolved** (*rule*). Whether at least one observed failure remains unrecovered at the end of the trajectory.
  - **Completion score** (*hybrid*). Whether the trajectory satisfies the completion-usability criterion derived from the combined rule-based and LLM completion scores.

Single-turn versus multi-turn indicators. Figure 16 compares indicator rates between single- and multi-turn trajectories. Multi-turn trajectories show higher action alignment (95.8% vs. 91.5%) and successful mutation (55.2% vs. 40.7%), with requirement evolution observed in 79.8% of eligible multi-turn trajectories. In contrast, completion outcomes are less favorable: final-answer consistency decreases from 53.0% to 26.0%, while unresolved failure increases from 23.5% to 30.3% and completion usability decreases from 17.0% to 7.7%. These results show that multi-turn trajectories contain more agent actions and requirement-related follow-up, but these behaviors do not correspond to higher completion quality. Because the cohorts are defined by the number of observed user messages, rather than the presence of substantive follow-up, the differences should be interpreted as associations with interaction depth rather than effects of multi-turn interaction itself.

Indicators by user-turn count. Figure 17 and Table 12 report trajectory-level indicator rates for trajectories with at least k user turns. Alignment remains consistently high across turn counts (92.9–96.9%), while successful mutation increases from 45.3% at $k = 1$ to 79.8% at $k = 10$. Requirement evolution rises sharply from 26.3% at $k = 1$ to 79.8% at $k = 2$ and exceeds 90% from $k = 4$ onward. In contrast, final-answer consistency decreases monotonically from 44.4% to 1.8%,

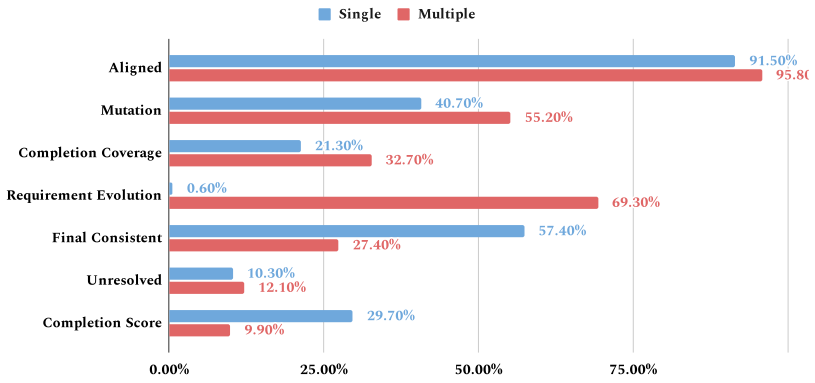


Figure 16: Trajectory-level indicator rates for single- and multi-turn cohorts. Completion coverage is computed over all trajectories in each cohort; requirement evolution, final consistency, unresolved failure, and completion usability are computed only among trajectories with completion-stage annotations.

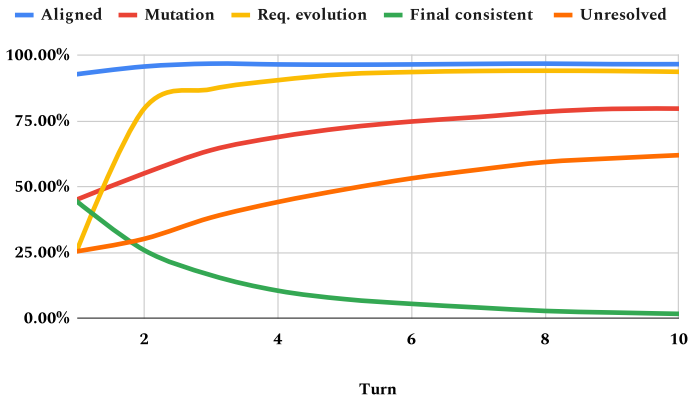


Figure 17: Trajectory-level indicator rates by user-turn count. For each k, rates are computed over trajectories with at least k user turns.

while unresolved failure increases from 25.6% to 62.1%. Thus, trajectories with more user turns exhibit more mutation and requirement-related follow-up but increasingly concentrate trajectories with weaker completion outcomes. These trends should not be interpreted as effects of additional turns: the cohorts are nested and progressively smaller, decreasing from 18,219 trajectories at $k = 1$ to 1,046 at $k = 10$. Moreover, mutation and requirement evolution are "ever observed" indicators with more opportunities to occur in longer trajectories, while successfully completed interactions may terminate earlier. User-turn count should therefore be interpreted as a descriptive trajectory characteristic rather than an independent measure of adaptation or quality.

Table 12: Indicator rates by user-turn count corresponding to Figure 17 (%).

Depth	Aligned	Mut.	Req. evo.	Final cons.	Unres.
1	92.9	45.3	26.3	44.4	25.6
2	95.8	55.2	79.8	26.0	30.3
3	96.9	64.0	87.2	16.5	38.4
4	96.6	69.0	90.6	10.6	44.3
5	96.5	72.5	92.9	7.4	49.1
6	96.6	74.9	93.7	5.6	53.3
7	96.8	76.6	94.1	4.2	56.6
8	96.9	78.6	94.2	2.9	59.5
9	96.7	79.7	94.1	2.3	60.9
10	96.7	79.8	93.8	1.8	62.1

Taken together, these results motivate separating *interaction value* from *supervision reliability* when selecting trajectories for training. Interaction value captures behaviors specific to realistic user–agent interaction, such as actionable follow-up, requirement evolution, correction, and recovery, whereas supervision reliability reflects whether the trajectory provides sufficiently grounded evidence for learning, including aligned actions, resolved execution, adequate verification, and evidence-consistent completion. This distinction supports a role-conditioned curation strategy: trajectories exhibiting both properties can serve as candidates for multi-turn SFT, while interaction-rich trajectories with unreliable outcomes may instead support repair, recovery, or preference supervision. Similarly, `unverifiable` trajectories can be routed to task reconstruction or further annotation rather than treated as agent failures.

B Benchmark Composition and Comparison

Table 13: Definitions of the benchmark-design dimensions used in Figure 6. A check mark is assigned only when the published benchmark protocol explicitly implements the corresponding property.

Code	**Dimension**	**Operational definition**
MT	Multi-turn interaction	The agent receives more than one user request within the same task episode.
PW	Persistent workspace	Repository and filesystem state produced by an earlier turn remains available to the agent in subsequent turns.
AF	Adaptive feedback	A later user message is generated from the interaction history or current task state, rather than replayed from a fixed, agent-independent script.
TD	Trajectory-derived tasks	Evaluation tasks are reconstructed from observed real user–agent interaction trajectories, rather than authored independently of such trajectories.
TE	Turn-level functional evaluation	Executable functional correctness is measured at intermediate interaction turns, allowing progress or regression across turns to be observed.
IQ	Interaction-quality metric	The benchmark reports a metric for interaction behavior or user experience in addition to executable task success.
EV	Executable verification	Task completion is determined using runnable tests or another programmatic verifier, rather than text-only judgment.

The feature coding in Figure 6 follows the published benchmark protocols (Jimenez et al., 2024; Wu et al., 2026b; Shen et al., 2026). The dimensions describe the default evaluation setup of each benchmark; a cross therefore indicates that the feature is absent from that setup, not that the benchmark could not be extended to support it.

Table 14: Detailed composition of the 95-task LEGO-UXBENCH development suite.

Statistic	Value	Statistic	Value
Task categories		*Instruction length (characters)*	
Application development	18	Maximum	1,722
Build and deployment	10	Minimum	17
Data analysis	15	Mean	460.9
Frontend development	29	Median	423
Machine learning	15		
UI optimization	8	*Checklist length (characters)*	
Programming languages		Maximum / minimum	1,023 / 208
JavaScript	44	Mean / median	433.5 / 403
Python	41		
Go	7	*Functional verification*	
SVG	3	Deterministic test cases	915
Manifest and evaluation		Tests per task (max / min)	19 / 3
Tasks	95	Tests per task (mean / median)	9.6 / 5
Checklist entries	569		
Scheduled attempts	190		

B.1 LEGO-UXBENCH

Using the task reconstruction pipeline introduced in Section 4, we construct LEGO-UXBENCH, an executable benchmark comprising 95 interactive coding tasks for multi-turn evaluation. The tasks are reconstructed from trajectories collected from CC-Bench Trajectories (Z.ai, 2025), GitHub PR Agent Trajectories (PotatoHD, 2026), and SWE-chat (SALT-NLP, 2026), and only candidates satisfying our executable evidence criteria are retained. Each task is materialized as an isolated Harbor case with a task instruction, initial workspace, execution environment, hidden functional tests, and a reference solution used to validate task construction. As summarized in Table 14, LEGO-UXBENCH spans six task categories and four programming languages, covering application development, build and deployment, data analysis, frontend development, machine learning, and UI optimization. The benchmark contains 915 deterministic test cases, with 3–19 tests per task, providing executable functional verification across heterogeneous coding settings.

B.2 EXISTING MULTI-TURN BENCHMARKS AND COMPARISON

SWE-Together and EvoCodeBench both evaluate coding agents in multi-turn interactions, but differ in task construction and interaction protocol. Figure 18 and Table 15 compare their task compositions by mapping tasks to a shared set of primary task types based on information provided by the original benchmarks. Shares are computed over tasks within each benchmark. In Figure 18, the inner rings show these primary task types, while the outer rings show difficulty levels for both benchmarks.

SWE-Together comprises 109 repository-level tasks reconstructed from real user–agent sessions and uses a reactive user simulator to provide feedback during task execution (Wu et al., 2026b). Under the harmonized taxonomy, 63.3% of its tasks involve bug fix and maintenance, 27.5% feature and implementation, and 9.2% refactor and migration. By difficulty, the benchmark contains 6 easy, 76 medium, and 27 hard tasks. EvoCodeBench comprises 26 stateful tasks and 227 interaction rounds, with persistent workspaces and executable tests under evolving requirements (Shen et al., 2026). Its tasks comprise feature and implementation (46.2%), verification and analysis (26.9%), bug fix and maintenance (19.2%), and refactor and migration (7.7%). The outer ring further shows the corresponding task subtypes, including greenfield development, integration, automation, brownfield development, security, migration, reproducibility, forensics, and explanation.

Comparison with LEGO-UXBENCH. LEGO-UXBENCH complements SWE-Together and EvoCodeBench primarily through its task composition. While SWE-Together focuses largely on bug fixing and maintenance and EvoCodeBench covers feature implementation and verification-oriented

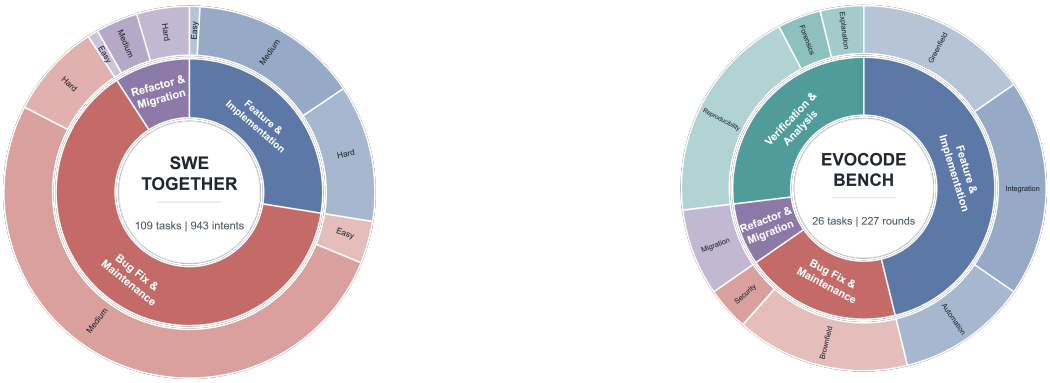


Figure 18: Task composition under the harmonized change-type taxonomy. **Left:** SWE-Together. **Right:** EvoCodeBench. Inner rings show primary task types; outer rings show difficulty levels for SWE-Together and task subtypes for EvoCodeBench.

Table 15: Task composition of SWE-Together and EvoCodeBench under the harmonized change-type taxonomy used in Figure 18.

Primary task type	**Outer-ring breakdown**	**Tasks**	**Share**
SWE-Together *(109 tasks; 943 intents)*			
Feature & Implementation	Easy: 1; Medium: 16; Hard: 13	30	27.5%
Bug Fix & Maintenance	Easy: 4; Medium: 56; Hard: 9	**69**	**63.3%**
Refactor & Migration	Easy: 1; Medium: 4; Hard: 5	10	9.2%
Total	Easy: 6; Medium: 76; Hard: 27	**109**	**100.0%**
EvoCodeBench *(26 tasks; 227 interaction rounds)*			
Feature & Implementation	Greenfield: 4; Integration: 5; Automation: 3	**12**	**46.2%**
Bug Fix & Maintenance	Brownfield: 4; Security: 1	5	19.2%
Refactor & Migration	Migration: 2	2	7.7%
Verification & Analysis	Reproducibility: 5; Forensics: 1; Explanation: 1	7	26.9%
Total	–	**26**	**100.0%**

tasks, LEGO-UXBENCH spans six diverse coding settings. Its design is motivated by our real-world trajectory analysis (Section 3.2), which reveals substantial diversity in user task types, execution surfaces, and technical topics. LEGO-UXBENCH thus provides a complementary evaluation setting that reflects the diversity of real-world user requests.

C ADDITIONAL EXPERIMENTAL DETAILS

C.1 TRAINING CONFIGURATION

Unless otherwise stated, all SFT and data-selection ablation runs use the same configuration in Table 16; only the selected training corpus changes.

Table 16: Training configuration.

Setting	Configuration
Initialization	Qwen3.5-35B-A3B-Base
Fine-tuning	Full-parameter supervised fine-tuning; with constant 1,000 selected trajectories per run
Input processing	`qwen3_5_nothink` template; 102,400-token cutoff; YaRN RoPE scaling
Optimization	Fused AdamW; 6 epochs; learning rate 5×10^{-5}; cosine schedule with 0.1 warmup ratio; weight decay 0.01; gradient clipping at 1.0
Batching	Per-device batch size 1; 4 gradient-accumulation steps; effective global batch size 64
Efficiency	FlashAttention-2, Liger kernels, and Unsloth gradient checkpointing

Table 17: Model-level values underlying Figure 7. Exact functional success is reported in percent; judge and interaction columns are normalized scores. The figure visualizes the full eleven-model cohort in Table 1; full dashboard results and metric definitions are available at coding-ux-eval.pages.dev.

Model	R1 exact	Final exact	Mean judge	Interaction proxy
GPT-5.6 Sol	68.4	68.4	0.703	0.755
Claude Opus 4.8	65.8	68.4	0.745	0.713
Claude Opus 4.7	64.7	67.4	0.733	0.714
MiMo-V2.6-Pro	63.2	65.8	0.716	0.705
GPT-5.5	64.2	65.8	0.750	0.748
Kimi K3	61.6	64.7	0.710	0.722
GLM-5.2	60.5	62.1	0.691	0.681
DeepSeek V4 Pro	59.0	60.5	0.718	0.735
Kimi K2.7	55.3	57.9	0.693	0.682
DeepSeek V4 Flash	52.6	57.4	0.631	0.645
DeepSeek V3.2	33.7	39.5	0.462	0.435

C.2 RECONSTRUCTION, ROLLOUT, AND EXPORT CONTRACTS

Table 18 specifies the artifact checked at each boundary. A source record is never discarded merely because a later stage cannot support its strongest intended use: where the intent and trajectory remain coherent, the record is demoted to the strongest supportable training-only tier. Unsafe, untraceable, or structurally invalid records are rejected with a machine-readable reason.

The fixed-round exporter performs six additional invariants before writing a JSONL row: (1) exactly five semantic user turns, excluding tool observations; (2) all assistant messages carry loss weight 1; (3) tool calls use the target OpenCode schema; (4) the expected reasoning block is singular, balanced, and in the required position; (5) task identifiers are unique across merged stages; and (6) no credential-like string is present. The exported row retains task, source, model, session, round, patch, and verifier identifiers so that it can be traced back to the raw Harbor trial. Raw trials and rejected rows remain outside the training shard for audit rather than being silently deleted.

Table 18: Artifact-level contracts in the SWE-Userfriendly–Harbor pipeline.

Boundary	Required evidence	Failure disposition
Source adapter	Lossless raw snapshot; normalized roles, tools, patch fields, and source provenance	Reject malformed/unsafe input; otherwise retain as `source_query` when executable fields are absent
Repository recovery	Resolved repository, pre-change commit, reference change, and aligned user-turn window	Demote coherent intent when replay is impossible; reject ambiguous or untraceable alignment
Environment build	Pinned environment builds and setup commands complete without infrastructure error	Record build, dependency, timeout, or container reason; do not treat it as a semantic failure
Executable admission	Original or generated verifier fails on base and passes with the reference change	Admit to matching fail-to-pass tier; otherwise demote or reject according to remaining evidence
Harbor materialization	Task, environment, tests, provenance, limits, commands, and hashes form a self-contained case	Reject incomplete cases before rollout; reference change remains hidden from the agent
Stateful rollout	One agent session/workspace; per-round instruction, trajectory, checkpoint, incremental/cumulative patch, and functional verifier	Record agent, API, timeout, and runtime failures separately; retries retain source identity
Strict SFT export	Five semantic user turns; continuous model/session; `final-original`=1; valid target tool and message schema	Reject from the strict shard with a structural or verifier reason; preserve raw trial artifacts

Table 19: Detailed composition of the protocol-aligned multi-source training mixture. Names describe source roles rather than internal experiment directories.

Source shard	Records	Role
Protocol-aligned round-1 synthetic trajectories	201	Controlled successful tool trajectories
Non-overlapping five-round synthetic trajectories	68	Explicit semantic multi-turn supervision
SWE-chat trajectories	444	In-the-wild user–agent interactions
ML-Marathon patch-to-bash trajectories	2	Repository patch execution
Potato-PR patch-to-bash trajectories	36	Repository patch execution
Accepted raw / converted for training	751 / 746	Five structurally invalid records rejected at conversion

Of the 746 converted records, 449 (60.2%) contain at least two semantic user turns; 297 contain one user request followed by a multi-step agent–tool trajectory. The converter normalizes 23,830 tool calls into 45,234 serialized messages. These production statistics document coverage and reproducibility; they are not treated as evidence of method effectiveness.

C.3 Additional transfer status

The reported SWE-bench Verified runs use OpenHands and 100 attempts per condition: pass@1 is 3.0% for the base and 26.0% for filtered SFT. Task-subset identity has not been verified, so these are descriptive scores rather than a paired capability-retention result. A matched comparison must reuse the same task manifest and selected base/SFT pair; evaluation under OpenCode and Claude Code is still needed.

The available 275-sample EvoCode-2403 run is a single-turn compatibility test, whereas EvoCode-Bench evaluates 26 persistent tasks over 227 rounds with MT@4 and SR (Shen et al., 2026). The current multi-turn run has incomplete or invalid fail-stop artifacts; no MT@4/SR result is inferred from the single-turn run.

C.4 Exploratory data-recipe comparisons

Table 20 collects the OpenCode results needed to compare training recipes. It deliberately excludes external benchmark columns: most intermediate checkpoints were not evaluated on those suites, and sparse cells would not form a valid comparison. These runs share the 95-task evaluation but are not a record-count scaling law; some interventions also change source mix, token exposure, and checkpoint selection.

Table 20: Representative training-data comparisons on LEGO-UXBENCH with OpenCode. All rows use 190 scheduled attempts. $\Delta = R_2 - R_1$. Rows are selected single-seed checkpoints; these comparisons are not controlled causal ablations.

Training condition	n	R_1	R_2	Δ	UX proxy	Errors
Base, no SFT	–	0.199	0.251	+0.052	0.249	8
Protocol-aligned mixture	746	0.303	0.376	+0.073	0.417	11
Recovery-oversampled mixture	801	0.245	0.325	+0.080	0.380	27
Closed-loop feedback, round-4 snapshot	884	0.302	0.355	+0.053	0.371	8
`retained27` variant	859	0.209	0.264	+0.055	0.287	8
Shuffled-feedback mixture	1,726	0.064	0.084	+0.020	0.089	5
Filtered interaction trajectories	11,383	**0.335**	**0.419**	**+0.084**	0.422	6
Harness-native trajectory cleaning	4,714	0.328	0.395	+0.066	**0.434**	23

These exploratory results do not isolate the effect of data volume. The 1,726-record shuffled-feedback mixture is substantially worse than the 746-record protocol-aligned mixture, while filtering 11,383 interaction trajectories gives the highest functional scores. Harness-native cleaning yields the highest UX proxy but more execution errors, showing that interaction quality and operational reliability must be reported separately.

The smaller interventions are diagnostic rather than isolated causal effects. Recovery oversampling increases $R_2 - R_1$ but lowers absolute performance, and the `retained27` variant is lower than its round-4 parent. A controlled follow-up would be a matched three-way comparison of native multi-turn, first-turn-only, and shuffled-feedback records at equal tasks, assistant-loss tokens, optimizer updates, and checkpoint rule. At least three training seeds and task-level bootstrap intervals are required for claim-bearing differences.

An earlier 746-record run also provides an exact end-to-end diagnostic: relative to its paired base run, round-1 exact success increases from 33/190 (17.4%) to 54/190 (28.4%), resolved-by-budget from 36/190 (18.9%) to 69/190 (36.3%), and attempts first resolved after round 1 from 3 to 15. These values describe a separate earlier run and should not be conflated with the filtered-SFT comparison in Table 20; they do not identify which curation component caused the gain.

Table 21: Ablation of the trajectory-depth constraint in Harddrop.

Model	LEGO-UXBENCH				SWE-Together		SWE-Bench Verified	Macro
	MeanIQ	Pass^2	Pass_1	Pass_F	Pass_1	Pass^2	Pass_1	Avg.
Harddrop	0.32	12.6%	18.4%	25.5%	4.6%	4.6%	39.0%	19.53
- Turn	0.28	11.6%	18.6%	25.0%	11.4%	4.6%	27.0%	18.31 (-1.22)

C.5 ABLATION ANALYSIS OF TRAJECTORY SELECTION

Ablation on hard filtering. Table 21 examines the trajectory-depth constraint in HardDrop. Removing it decreases the macro average from 19.53 to 18.31, a drop of 1.22 points. The effect is benchmark dependent: it slightly improves LEGO-UXBENCH Pass_1 (18.4% to 18.6%) and SWE-Together Pass_1 (4.6% to 11.4%), but lowers LEGO-UXBENCH MeanIQ (0.32 to 0.28), final success (25.5% to 25.0%), and SWE-bench Verified Pass_1 (39.0% to 27.0%). Overall, the depth constraint improves aggregate performance despite these task-dependent trade-offs.

D LIMITATIONS

Our evaluation approximates coding-agent user experience through executable verification, an LLM-simulated user, and an LLM-based interaction-quality judge. These model-based components enable controlled, stateful evaluation but do not directly measure human preferences. Simulator behavior may depend on the chosen model and prompt, and adaptive feedback can introduce variation across rollouts. The 95-task benchmark and two-rollout protocol also limit the precision with which small differences between models can be interpreted.

The trajectory analyses are observational and reflect the composition and annotation coverage of the available source corpora. Associations with interaction depth therefore do not establish causal effects of additional user turns. Likewise, the training comparisons use selected single checkpoints and may differ in data composition, token exposure, and checkpoint selection; the reported macro averages summarize the displayed metrics rather than isolating the effect of an individual curation rule.

Finally, fail-to-pass validation establishes an executable contrast between the recovered base state and reference change, but it does not guarantee complete semantic coverage of the original user intent. This limitation is especially relevant when tests are generated from task descriptions. Broader human audit, mutation-based test analysis, additional training seeds, and evaluation on more tasks and interaction protocols would strengthen the corresponding validity and generalization claims.