TRACE2EVAL: CONTEXTUALIZED EVALUATION OF CODING AGENTS FROM REAL-WORLD TRACES

Anonymous authors
Paper under double-blind review

ABSTRACT

Coding agents are increasingly used in real-world workflows across diverse domains, motivating evaluation grounded in real user tasks. Existing benchmarks, however, largely rely on static public tasks with reference patches or hidden tests. Next-generation coding-agent benchmarks should therefore move beyond correctness on fixed tasks toward continually renewed evaluation of overall artifact quality in real user workflows. We propose Trace2Eval, a dynamically updatable pipeline that constructs benchmarks from real-world interaction trajectories. Trace2Eval employs a multi-agent reconstruction system to analyze the filtered trajectory, recover the original user query, restore the workspace files to their pre-interaction state, and rebuild a trajectory-equivalent execution environment. A candidate agent then executes the reconstructed query to produce an initial artifact. To evaluate these artifacts, the pipeline iteratively refines human rubrics through inter-annotator alignment, and calibrates judge agents against human assessments using source-code and runtime evidence. Guided by feedback from the Trace2Eval pipeline, models revise their initial artifacts, yielding a mean score gain of 10.05 percentage points across 19 model-harness configurations and a maximum of 19.28 percentage points for DeepSeek-V4-Pro with Claude Code, which supports rubric feedback as actionable guidance for artifact improvement.[1]

1 INTRODUCTION

Powered by increasingly capable frontier models (OpenAI, 2026c; Anthropic, 2026), coding agents such as Codex (OpenAI, 2025) and Claude Code (Anthropic, 2025) can inspect and edit repositories, execute commands, and complete tasks in users' workspaces. Benchmarks evolved from evaluating isolated code to assessing changes within existing software projects. HumanEval and MBPP assessed function-level solutions through tests (Chen et al., 2021; Austin et al., 2021). APPS introduced competition-style problems (Hendrycks et al., 2021), while DS-1000 and BigCodeBench expanded coverage to library-rich tasks (Lai et al., 2022; Zhuo et al., 2024). LiveCodeBench added time-sliced problems to renew evaluation as models evolved (Jain et al., 2024). SWE-bench moved to repository-level issue resolution, where agents worked from repository snapshots and produced patches assessed against test outcomes (Jimenez et al., 2023).

Although this evolution brought evaluation closer to real development through more demanding repository-level tasks, important gaps remain between benchmark settings and real user workflows, raising several challenges for reliable evaluation. Public programming problems and repository issues make tasks and reference solutions increasingly retrievable, leaving static collections vulnerable to contamination and benchmark hacking (OpenAI, 2026a; Wang et al., 2026a; Oren et al., 2023). Time-sliced evaluation renewed task content (Jain et al., 2024), but freshness alone does not establish whether the selected activities match evolving user demand: competition problems and issue-to-patch repair cover only part of the work users delegate to coding agents. Even for similar requests, a benchmark's prescribed repository snapshot may differ from the user's initial files and dependencies, changing the work needed to complete the task. Correctness tests then assess behavior encoded in their checks, while a delivered application can implement tested operations yet omit the controls needed to use them. Such an artifact may satisfy component-level checks without completing the

[1] Supporting materials are available at https://anonymous.4open.science/r/TRACE2EVAL-ICLR.

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.

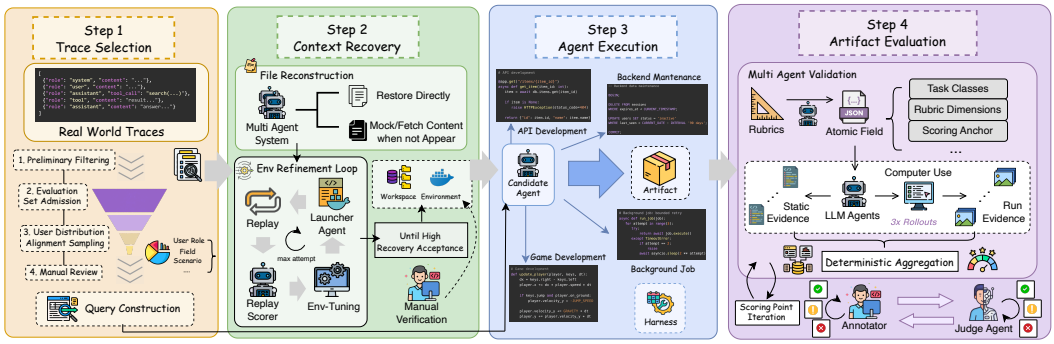


Figure 1: Overview of the Trace2Eval workflow. **(1) Trace selection.** Two-stage greedy selection aligns candidates with online request profiles, then combines rubric-based admission with distribution constraints. **(2) Context recovery.** For each retained request, recovery agents reconstruct its initial workspace and environment, using trajectory replay to assess and refine the recovered context. **(3) Agent execution.** Candidate coding agents execute the request in this context and produce an artifact. **(4) Artifact evaluation.** Judge agents assess the artifact using source and runtime evidence and human-grounded rubrics. Repeated scoring yields dimension-level and aggregate quality scores.

user's workflow. Connecting scores to real use therefore requires representative requests, their original context, and criteria for the quality of the delivered artifact.

These observations motivate contextualized evaluation, which assesses the requests people delegate to coding agents within their working environments, including the initial files, dependencies, and delivery requirements. A sustainable benchmark for this setting should satisfy four properties at once. **(1) Authenticity.** Tasks should come from real user traffic, preserving the original goals and context that distinguish practical workflows from curated problems. **(2) Contamination resistance.** Evaluation should avoid reliance on publicly retrievable solutions, assessing delivered artifacts against user requirements rather than an available reference patch. **(3) Dynamism.** A refreshable construction procedure should periodically re-sample user traffic and re-score artifacts as users' goals, workflows, and agent capabilities evolve. **(4) Measurability.** Explicit scoring criteria should translate user-perceived quality into measurable judgments, making scores interpretable even when the original request is underspecified. No existing benchmark meets all four requirements. Meeting these requirements calls for a reusable evaluation protocol that follows evolving user needs while maintaining consistent standards for artifact quality.

We propose Trace2Eval, a reusable evaluation protocol implemented as a dynamic pipeline that constructs coding-agent benchmarks from real-world traces (Figure 1). Given an anonymized interaction trace τ between a user and a coding agent, Trace2Eval reconstructs a benchmark instance $x = \langle q, \widehat{W}_0, \widehat{E}_0 \rangle$, where q is the user task distilled from the trace, $\widehat{W}_0$ is the recovered workspace before the recorded interaction, and $\widehat{E}_0$ is a containerized approximation of the user's initial execution environment. A candidate agent runs from x to produce an artifact y, which judge agents assess against human-grounded rubrics. This accommodates open-ended tasks without requiring a unique reference patch. To further assess the rubric's validity, we translate diagnosed weaknesses into revision feedback and reassess the resulting artifacts. Verified repairs and score gains provide complementary evidence that the rubric captures meaningful aspects of user-perceived quality.

Our contributions are as follows:

- **A dynamic, trace-driven evaluation paradigm.** We introduce evaluation-as-construction: rather than a static dataset, Trace2Eval is a periodically executable pipeline that re-samples real online traffic and reconstructs $\langle q, \widehat{W}_0, \widehat{E}_0 \rangle$ instances, so the evaluation tracks both evolution of user intent and model capability.
- **Recovery of initial task contexts from incomplete traces.** We develop a reconstruction workflow that combines observed file evidence, contextual completion, and dependency inference to recover the initial workspace and environment. Bounded replay and refinement assess whether the recovered context supports the recorded tool behavior.

- **Rubric-based quantification of user experience.** We use rubrics to translate user-perceived delivery quality into explicit, dimension-specific scoring criteria, together with a standardized procedure for constructing ground truth and aligning judge agents with human evaluations.
- **Evaluation and rubric-guided artifact improvement.** To test whether rubric-based assessment provides useful improvement signals, we evaluate initial artifacts, turn diagnosed weaknesses into revision instructions, and reassess the revised artifacts. Total scores rise by 10.05 pp on average, with DeepSeek-V4-Pro gaining 19.28 pp under Claude Code (Table 1), providing complementary evidence for the practical value of our rubric and scoring framework.

2 Related Work

Static code and repository-level evaluation. Repository-level evaluation expanded from issue resolution in SWE-bench and its extensions (Jimenez et al., 2023; OpenAI, 2024; Deng et al., 2025) to project-level code generation in DevEval (Li et al., 2024) and repository construction in NL2Repo-Bench (Ding et al., 2025). Although this progression broadened task scope, predefined tasks and correctness criteria still capture only part of what users expect from coding agents. Public benchmark exposure further threatens evaluation validity (OpenAI, 2026a), while changing user goals make fixed task distributions less representative over time. These limitations motivate Trace2Eval's use of fresh user traces, which preserve the context of actual requests and support evaluation of delivered artifacts as usage evolves.

Environment construction. Environment construction supports scalable task generation through several routes. SWE-smith created test-breaking tasks in repository environments (Yang et al., 2025), whereas SWE-Mirror transferred issue semantics into repositories with configured environments (Wang et al., 2025). These approaches expanded task collections by exploiting existing code and tests, reducing the need to configure each task independently. For terminal tasks, TerminalTraj and TermiGen constructed Docker environments for trajectory synthesis (Wu et al., 2026; Zhu et al., 2026), while CLI-Universe combined taxonomy-guided task generation with executable verification (Hua et al., 2026). TerminalWorld reconstructed terminal tasks from public recordings and evaluated completion with executable tests (Chu et al., 2026). Trace2Eval instead recovers the initial contexts of user–agent interactions and assesses delivered coding artifacts across functionality, code quality, and user experience with human-grounded rubrics.

Rubrics, judges, and feedback. Open-ended evaluation combines explicit quality criteria with judgments whose reliability must be established. HELM advocated measuring multiple dimensions (Liang et al., 2022), while rubric research studied how such dimensions become criteria for evaluation and training (Liu et al., 2026). G-Eval used structured criteria to improve agreement with human assessments (Liu et al., 2023); MT-Bench examined both the agreement and biases of LLM judges (Zheng et al., 2023), and AlpacaEval provided a human-validated automatic evaluator (Li et al., 2023). Together, these studies motivate checking automated scores against human judgments rather than assuming that explicit criteria alone ensure reliability. Beyond scoring, Self-Refine used feedback to revise generated outputs (Madaan et al., 2023); Asuka-Bench and CATArena studied feedback-driven development and iterative code improvement (Wang et al., 2026b; Fu et al., 2025), connecting assessment with improvement. Trace2Eval uses judge agents to automatically score coding artifacts against human-grounded rubrics, drawing on source-code and runtime evidence.

3 Trace2Eval

Starting from anonymized real-world user traces, Trace2Eval builds executable evaluation instances for coding agents and scores their artifacts with multi-dimensional rubrics rather than a single reference patch. As shown in Figure 1, the pipeline first retrieves and labels user traces (§3.1), then recovers an evaluation instance $\langle q, \widehat{W}_0, \widehat{E}_0 \rangle$, including the user query q, recovered workspace $\widehat{W}_0$, and recovered execution environment $\widehat{E}_0$ (§3.2). The evaluated model is then run from the recovered instance to produce an artifact, which is assessed with human-grounded rubrics and a static-dynamic LLM-as-judge framework (§3.3). This design evaluates agents in the original user context rather than a curated GitHub-issue setting. By periodically selecting new online traces and updating the dataset, Trace2Eval keeps evaluation aligned with evolving real-world usage.

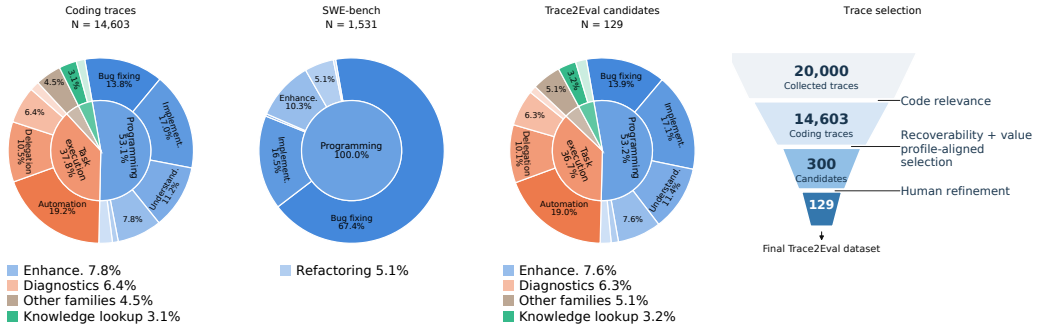


Figure 2: Task-type distributions and trace selection. The pies compare online coding traces, SWE-bench, and selected candidates; inner disks show broad task families (`x1`), and outer rings show finer activities (`x2`). Candidates retain the broader mix of online requests, whereas SWE-bench concentrates on programming and bug fixing. The funnel shows the successive selection stages leading to the final benchmark.

3.1 Retrieving, Filtering, and Labeling Online Traces

We collect 20,000 anonymized traces from online coding-agent traffic between June and July 2026 by randomly sampling approximately equal numbers per day, of which 14,603 are identified as code-related and form the population for our distribution analysis. We label coding traces using the 171-category, six-axis taxonomy summarized in Appendix A.6. Its four top-level dimensions are task type (`x1`/`x2`), usage scenario (`y`), technical domain (`z1`/`z2`), and user role (`w`). A two-stage greedy procedure first selects 300 distribution-aligned candidates (Algorithm 1), then combines admission-rubric voting with distribution constraints to select the final 129 tasks (Table 4).

Figure 2 compares the task-type distributions of online coding traces, SWE-bench, and the final 129-task Trace2Eval benchmark, alongside the selection stages. The visualization shows that the selected candidates cover common activities in online coding requests, whereas SWE-bench concentrates more heavily on debugging. This broader coverage connects evaluation to the range of work users delegate to coding agents. Appendix A.1 extends the comparison across all six profile sub-axes. The second greedy stage preserves distribution alignment while admitting tasks that meaningfully distinguish coding-agent capabilities (Appendix A.1.3).

3.2 Workspace and Environment Recovery

Online traces provide tool observations rather than complete snapshots of the user's workspace and environment. A complete ground-truth state is therefore unavailable for direct comparison, so we assess recovery through the behavior of the reconstructed context. We use *trajectory equivalence* as the joint acceptance criterion for the recovered workspace and environment: whether replaying the original tool calls reproduces their recorded results.

Workspace and environment recovery. Let W_0 and E_0 denote the original initial workspace and environment, with hats marking recovered estimates. We reconstruct $\widehat{W}_0$ using file evidence from the trace, restoring directly observed content and filling gaps through contextual inference or retrieval from verified public sources. The recovered workspace $\widehat{W}_0$ excludes artifacts introduced during the session and restores modified pre-existing files to their initial state. Environment recovery similarly separates pre-existing dependencies from changes made during execution. Let $\tau = (T_1, \ldots, T_T)$ be a trace of T tool calls, with $T_t = (u_t, i_t, o_t)$ denoting the tool name, input, and recorded output at step t. We infer the initial environment E_0 and each step's environment change ΔE_t through the update

$$\mathrm{Agent}\big(C_{E_t}, C_{E_0}^{(t)} \mid T_t\big) = \big(C_{E_{t+1}}, C_{E_0}^{(t+1)}\big), \tag{1}$$

where Agent is the recovery agent operating on two container states: C_{E_t} is the current replay environment before call t, while $C_{E_0}^{(t)}$ is the evolving estimate of the initial environment after the preceding calls. Within each replay round, call outcomes guide updates to both states, distinguishing

initial dependencies from later changes. The resulting initial-state container realizes $\widehat{E}_0$, while the inferred stepwise changes define $\widehat{\Delta E}_t$. Multiple replay rounds refine both states before we package $\widehat{E}_0$ as the evaluation image. Appendix A.2 details the file-recovery agents and environment updates.

Trajectory equivalence through replay. We extract tool calls from the original trace and replay them in order in the recovered workspace and cold-started environment. For each call, a replay evaluation agent compares the replay result a_t with the recorded result o_t, using the tool and input as context, and assigns

$$m_t = \mathrm{Judge}_{\mathrm{replay}}(u_t, i_t; a_t, o_t) \in [0, 1]. \tag{2}$$

The score m_t comes from the replay evaluation agent's judgment of agreement: it assigns 1 when the results are identical and lower scores for more substantial differences. For each replay round, let $\mathbb{I}(\tau)$ contain the indices of calls with scored comparisons. For $|\mathbb{I}(\tau)| > 0$, the *replay score* quantifies trajectory equivalence as the arithmetic mean of these per-call scores:

$$\mathrm{ReplayScore}(\widehat{W}_0, \widehat{E}_0; \tau) = \frac{1}{|\mathbb{I}(\tau)|} \sum_{t \in \mathbb{I}(\tau)} m_t. \tag{3}$$

When all tool calls in τ are replayed and scored, a replay score of 1 indicates that the recovered workspace and environment reproduce the original results throughout the trajectory. This trajectory-equivalent workspace and environment form our recovery target, with the replay score providing the quantitative basis for recovery acceptance. Recovery agents repair the file or environment defects revealed by low-scoring calls, then replay the trajectory from the revised context to verify the corrections. This cycle continues until the discrepancies are resolved or the replay budget is reached, followed by human verification (Appendix A.2).

3.3 Rubric Construction Protocol and LLM-as-Judge Scoring

We assess artifacts produced in the recovered task contexts using a human-grounded rubric, combining expert design with calibrated agent judgment.

Expert-guided rubric construction. A qualitative analysis of human preference rationales identifies the quality concerns that inform the rubric (Appendix A.3.1). Experts review their relevance and scope, confirm 11 dimensions across accuracy, code quality, and user experience, and define initial anchors for scores $h_{i,d} \in \{0, 1, 2\}$ for artifact i on dimension d. They set group weights of 50%, 35%, and 15%, respectively, and within-group priorities based on expert judgment of their importance for user-perceived quality (Table 8). This specification provides the starting point for subsequent human and agent calibration.

Static-Dynamic LLM-as-judge scoring. Each rubric dimension specifies the evidence required for scoring. Seven dimensions are assessed from source code, while code execution, data authenticity, interface and interaction, and information presentation require runtime or rendered evidence. Static analysis and runtime agents collect evidence for the judge agent, implemented by GLM-5.3-Flash in the model comparison. For each instance i and dimension d, the scoring workflow performs K independent evidence-collection rollouts, indexed by k. Let $e_{i,d}^{(k)}$ be the evidence collected in rollout k, and let R_d be the rubric anchor for dimension d. The judge agent J maps evidence to a discrete rubric score,

$$\hat{h}_{i,d}^{(k)} \;=\; J\big(R_d,\, e_{i,d}^{(k)}\big) \;\in\; \{0, 1, 2\}. \tag{4}$$

A deterministic aggregator combines the K rollout scores into the final score, consensus label, and self-consistency statistics (Appendix A.3.2).

Rubric-level ground truth. Since we need a reference score for each artifact to iteratively refine the rubric definitions and score anchors, we define rubric-level ground truth as the adjudicated human score for each artifact–dimension pair, denoted by $g_{i,d} \in [0, 2]$. The first two human annotation rounds provide independent scores and evidence-based rationales for each artifact. In the third round, a third annotator reviews both sets of scores and rationales alongside the artifact itself, then assigns the final reference score $g_{i,d}$ (Appendix A.3.1).

Calibration proceeds in two stages. We first refine the human score anchors to achieve high human-human agreement and establish the adjudicated reference $g_{i,d}$. We then give the judge agent this human rubric as its initial scoring instructions. Because its scores do not align sufficiently with the human reference, we refine the agent scoring points over multiple rounds while keeping the dimensions, weights, and human reference fixed. The resulting agent anchors specify evidence requirements and decision boundaries learned from these disagreements. With these refinements, judge agents reach human-level agreement on several dimensions, supporting scalable rubric-based evaluation (detailed in Appendices A.3 and A.4.3).

4 EXPERIMENTS

We organize the evaluation around three research questions:

- **RQ1:** How faithfully can Trace2Eval recover the initial workspace and environment from user traces?
- **RQ2:** How accurate is the evaluation? Does the rubric reflect user-perceived quality (RQ2.1), and do judge agents agree with human ground truth (RQ2.2)?
- **RQ3:** Can the evaluation guide agent improvement? What capabilities do models exhibit across harnesses (RQ3.1), and does rubric-derived feedback improve their artifacts (RQ3.2)?

4.1 EXPERIMENTAL SETUP

Candidate models with specific harness first execute the 129 Trace2Eval tasks in their recovered contexts through Claude Code or OpenHands within Terminal-Bench (Merrill et al., 2026). GLM-5.3-Flash evaluates the resulting artifacts through three scoring rollouts under the calibrated rubric. GLM-5.2 diagnosis, design, and review agents turn rubric-derived feedback into revision queries; each model then improves its artifact using the same harness and preserved task environment. Re-evaluation under the same rubric tests whether the assessment provides useful guidance for improvement. Appendix A.4 documents execution and component validation, while Appendix A.5 details feedback preparation and revision.

4.2 RECOVERY FIDELITY (RQ1)

Each instance $\langle q, \widehat{W}_0, \widehat{E}_0 \rangle$ uses a retained user request q, with initial workspace $\widehat{W}_0$ and environment $\widehat{E}_0$ reconstructed from file and dependency evidence in its trace. To validate the usability of the recovered workspace and environment, human reviewers inspect the initial files for completeness, while replay tests tool behavior in the cold-started environment. The replay evaluation agent compares tool-call results with the original observations (§3.2).

On the recovery calibration set, we compare replay fidelity before and after refinement. The difficulty-stratified analysis shows larger gains where startup repair leaves behavioral discrepancies, indicating that replay identifies defects beyond those exposed at launch. Appendices A.4.1–A.4.2 detail the comparisons. With an 82.6% workspace acceptance rate and a mean replay score of 0.9175, these results answer RQ1 by demonstrating that recovery can provide high-quality initial workspaces and environments for executable agent evaluation grounded in real users' development contexts.

4.3 EVALUATION ACCURACY (RQ2)

Alignment with user-perceived quality (RQ2.1). To ground artifact evaluation in real development needs, we collect expert judgments of the relative importance of the quality concerns identified in human preference rationales (Appendix A.3.1). These judgments determine the final group weights and within-group priorities in Table 8.

For example, the final weights of 22.5% for requirement implementation and 20.0% for code execution preserve the experts' relative priorities. This alignment emphasizes completing the user's request and delivering a runnable artifact, reflecting expert perceptions of user needs and addressing RQ2.1.

Human agreement (RQ2.2). To test whether this human-grounded rubric supports automated evaluation, we compare judge agents with adjudicated human ground truth (GT; §3.3). Figure 11,

Table 1: Artifact quality before (R1) and after rubric-guided revision (R2) on the full 129-task Trace2Eval benchmark. Scores use % of maximum; changes are in percentage points (pp). Acc, Code, and UX denote accuracy, code quality, and user experience. Within each harness and column, bold and underlining mark the best and second-best values. Model names link to official documentation, cited in Appendix A.4.6.

Model	Acc		Code		UX		Total		Change (pp)			
	R1	R2	R1	R2	R1	R2	R1	R2	ΔAcc	ΔCode	ΔUX	ΔTotal
Claude Code												
DeepSeek-V4-Pro	75.10	97.19	82.40	93.14	66.30	89.50	76.85	96.13	**+22.09**	+10.74	+23.20	**+19.28**
GLM-5.2	79.80	94.80	86.90	96.46	68.40	88.01	80.96	94.13	+15.00	+9.56	+19.61	+13.17
GLM-5.3	83.50	97.13	90.70	99.18	70.10	**93.43**	84.76	97.29	+13.63	+8.48	**+23.33**	+12.53
GPT-5.5	76.80	89.04	86.70	95.83	59.40	78.24	78.10	89.88	+12.24	+9.13	+18.84	+11.78
GPT-5.6 sol	85.93	94.37	92.44	97.36	74.82	88.47	86.23	94.87	+8.44	+4.92	+13.65	+8.64
Kimi-K3	78.43	88.27	87.92	93.86	60.24	76.49	78.93	88.07	+9.84	+5.94	+16.25	+9.14
MiniMax-M3	75.80	90.85	85.40	96.24	58.90	82.20	76.64	91.57	+15.05	**+10.84**	+23.30	+14.93
LongCat-2.0	82.20	90.94	90.85	96.13	64.40	79.81	82.65	92.04	+8.74	+5.28	+15.41	+9.39
Fable 5	**92.31**	**97.26**	**96.05**	**99.28**	**81.76**	90.93	**92.42**	**97.37**	+4.95	+3.23	+9.17	+4.95
OpenHands												
DeepSeek-V4-Pro	87.85	92.13	92.01	97.59	75.00	89.89	85.72	93.75	+4.28	+5.58	+14.89	+8.03
Gemini 3.6 Flash	82.56	91.02	90.15	96.10	70.68	86.24	82.89	91.76	+8.46	+5.95	+15.56	+8.87
GLM-5.2	86.90	94.24	93.03	**98.67**	67.52	88.53	84.27	95.09	+7.34	+5.64	+21.01	+10.82
GLM-5.3	82.10	90.43	90.19	95.10	72.07	78.55	84.90	90.41	+8.33	+4.91	+6.48	+5.51
GPT-5.5	84.11	93.30	90.13	96.03	65.47	88.33	82.86	94.62	+9.19	+5.90	**+22.86**	**+11.76**
GPT-5.6 sol	88.23	**95.36**	**93.54**	98.33	**78.14**	**93.38**	87.46	95.89	+7.13	+4.79	+15.24	+8.43
Kimi-K3	77.04	86.67	86.84	92.76	59.43	76.27	77.54	86.16	**+9.63**	+5.92	+16.84	+8.62
MiniMax-M3	84.27	90.08	90.67	98.05	65.90	86.07	82.40	91.97	+5.81	**+7.38**	+20.17	+9.57
LongCat-2.0	78.72	88.18	88.21	93.96	61.43	78.39	79.52	88.18	+9.46	+5.75	+16.96	+8.66
Fable 5	**88.87**	95.12	93.06	97.58	76.57	93.11	**88.98**	**95.91**	+6.25	+4.52	+16.54	+6.93

Table 13, and Appendix A.4.3 provide the criterion-level evidence. The comparisons distinguish usability agreement, based on zero versus nonzero grades, from exact agreement across quality levels. Cohen's $\kappa = (p_o - p_e)/(1 - p_e)$ adjusts observed agreement p_o for agreement p_e expected by chance from the grade distributions.

The judge agent achieves human-comparable or higher exact agreement on selected dimensions, such as architecture. Feature richness also shows an advantage over the human average in both exact agreement and κ. Overall usability agreement of 93.0%, compared with 92.9% for humans, validates the judge agent's accuracy in identifying artifact usability and supports its use as a partial substitute for human assessment under the rubric, answering RQ2.2.

4.4 Agent Performance and Feedback (RQ3)

We compare initial delivery and feedback-guided revision to examine model capabilities and validate the rubric's guidance. R1 denotes initial execution; R2 revises the resulting artifact using rubric feedback, with the same model, harness, and preserved environment.

Model capability profiles (RQ3.1). We first examine whether stronger code quality also yields better user experience in R1 (Table 1). Under OpenHands, GLM-5.2 leads DeepSeek-V4-Pro in code quality but trails it by 7.48 pp in user experience and also has a lower Total score. Under Claude Code, LongCat-2.0 and GLM-5.3 have nearly identical code-quality scores (90.85 versus 90.70), yet GLM-5.3 leads by 5.70 pp in user experience. Figure 3 adds execution cost to these profiles. Across both harnesses, these comparisons show why code quality alone cannot characterize practical delivery: Trace2Eval distinguishes models by how well their artifacts serve users, answering RQ3.1.

Feedback reveals different model strengths (RQ3.2). Beyond agreement with human judgments, we test whether the rubric's diagnoses help models improve their artifacts. Under Claude Code, DeepSeek-V4-Pro shows strong repair ability despite weak initial delivery: its Total rises from 76.85

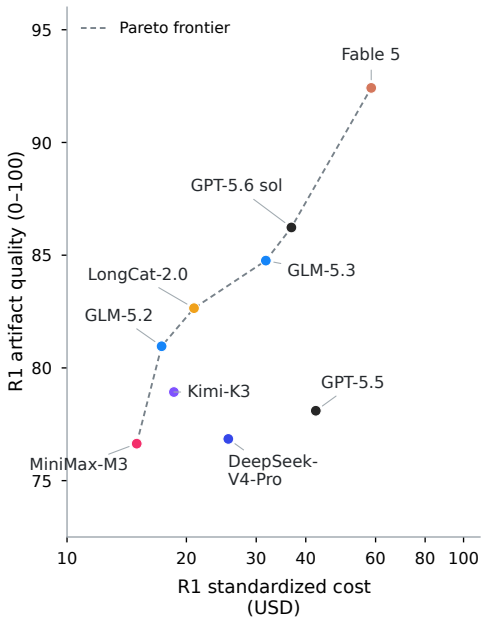


Figure 3: R1 artifact quality versus standardized execution cost under Claude Code. The dashed line connects Pareto-efficient models.

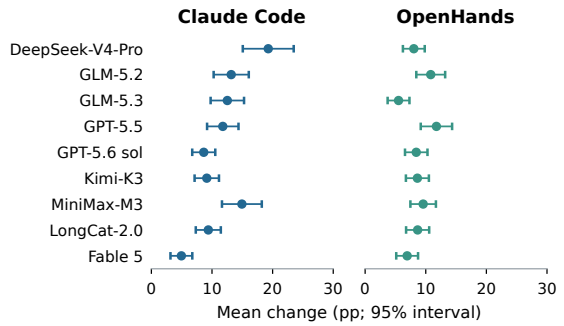


Figure 4: Mean R2–R1 Total gains (pp) under each harness. Bars show paired-task bootstrap 95% CI.

Table 2: Task-level outcomes on all 129 Trace2Eval tasks. Imp., Same, and Reg. denote increased, unchanged, and decreased Total scores.

	Claude Code			OpenHands		
Model	Imp.↑	Same↓	Reg.↓	Imp.↑	Same↓	Reg.↓
DeepSeek-V4-Pro	70	28	31	85	23	21
GLM-5.2	84	15	30	106	11	12
GLM-5.3	83	27	**19**	94	**10**	25
GPT-5.5	**95**	**11**	23	**108**	12	**9**
GPT-5.6 sol	83	13	33	91	11	27
Kimi-K3	84	13	32	88	11	30
MiniMax-M3	89	15	25	97	14	18
LongCat-2.0	88	13	28	87	12	30
Fable 5	71	24	34	86	13	30

to 96.13 (+19.28 pp), the largest gain in Table 1. This response indicates that initial task completion alone does not capture how effectively a model can use explicit quality feedback. Fable 5 presents a complementary strength: it delivers the strongest initial artifacts (92.42) and refines them to 97.37, nearing the scoring ceiling. Feedback use also depends on the harness: GLM-5.3 starts at nearly identical Total scores under Claude Code and OpenHands (84.76 versus 84.90), but finishes 6.88 pp higher under Claude Code after revision. Rubric feedback thus supports both substantial repair and further refinement of already strong artifacts, answering RQ3.2 (Figure 4; Table 2).

Feedback ablation. To determine whether R2 improves because of rubric feedback or another development opportunity, we construct an unguided condition for each model. It starts from the model's R1 artifact and environment and repeats the original prompt unchanged. The guided condition uses the same starting state with rubric-derived repair instructions. Figure 5 and Table 16 compare both against R1. The smaller unguided gains suggest that another development pass helps, but targeted diagnoses provide substantial additional benefit. This comparison supports the answer to RQ3.2 by linking rubric feedback to artifact improvement (Appendix A.4.7).

Case studies: Feedback-guided artifact repair. We inspect concrete repairs to test whether score gains correspond to more usable workflows. In a workflow-management task, DeepSeek-V4-Pro initially implements backend APIs and client-side calls but leaves a placeholder frontend. Rubric feedback identifies the missing interface; the model adds pages, forms, and navigation. Browser evidence verifies these interactions, and interaction and information-presentation grades both rise from 0 to 2. This repair turns implemented components into functionality users can operate (Appendix A.5.2). Appendix A.5.3 traces a complementary repair from a failed build to a working login service.

The MiniMax-M3 commerce case (Appendix A.5.2) illustrates incomplete repair: feedback leads to new backend endpoints, but inconsistent frontend routes and requests still interrupt the workflow, leaving code execution at grade 1. Gains thus depend on how fully the model carries a diagnosed

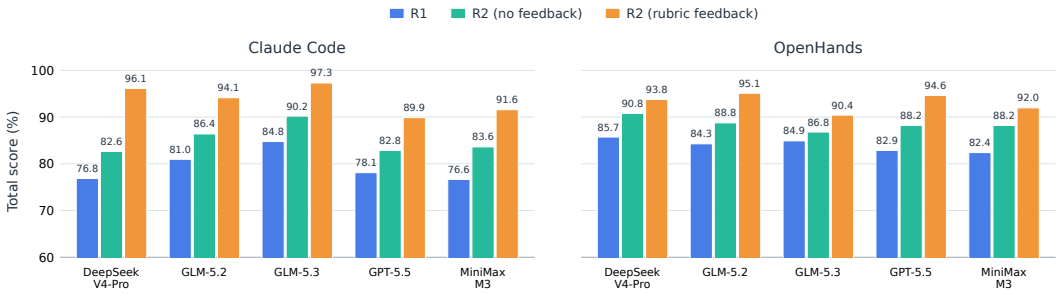


Figure 5: Total scores for five representative models under Claude Code and OpenHands. Both revisions begin with the same R1 artifact: one uses only the original request, while the other receives rubric feedback. The bars compare R1, unguided R2, and rubric-guided R2 using complete experimental results, with rubric-guided scores also reported in Table 1.

repair through related components. Checking end-to-end usability lets the rubric distinguish partial edits from completed repairs.

The model-specific improvements and verified workflow repairs show that Trace2Eval provides actionable feedback while distinguishing initial delivery, responsiveness to diagnosis, and the ability to complete a repair. This supports the rubric's practical validity and answers RQ3.2.

5 Discussion

Contextualized evaluation and actionable feedback. Recovery enables evaluation in contexts drawn from real requests, while replay validates the recorded behavior (§4.2). Two-stage greedy selection combines distribution alignment with rubric-based admission to retain meaningful evaluation tasks (Appendix A.1). Within these contexts, code quality and user experience reveal complementary aspects of delivery. The feedback ablation links dimension-level diagnosis to artifact improvement beyond an unguided revision (§4.4), showing how evaluation can also guide development. The qualitative cases suggest that guidance is most useful when it identifies a concrete defect and a verifiable repair. Benefits depend on whether the candidate carries that repair through the affected components, explaining the coexistence of average gains and task-level regressions.

Limitations. Although judge agents achieve agreement comparable to human annotators on several dimensions, human–agent agreement still leaves room for improvement. We are refining the judge agents to improve consistency with human assessments. In addition, the file and environment recovery workflow occasionally produces task packages that lack core files even after human verification. Such omissions can obstruct subsequent agent execution, highlighting the need for further checks on the completeness of recovered workspaces.

6 Conclusion

We present Trace2Eval, a refreshable procedure that turns real coding-agent traces into contextualized evaluation tasks. By recovering the initial workspace and environment, validating observed behavior through replay, and assessing artifacts with human-grounded rubrics, it connects the circumstances of a user request to explicit criteria for delivery quality. The component studies establish the feasibility and current limits of this construction, while evaluation on the 129-task benchmark reveals differences between code quality and user experience that motivate dimension-level assessment. Together, they position trace-driven construction as a basis for evaluation that can be renewed as user needs evolve, with each version's coverage and measurement reliability assessed explicitly.

AI USE STATEMENT

Generative AI supports three parts of this work. For *benchmark construction*, Claude Sonnet 4.6 labels traces and votes on admission. Claude Opus 4.6 powers the restorer, mocker, fetcher, and validator agents for file recovery, together with the environment recovery, tuning, launcher, and replay evaluation agents (Appendices A.1 and A.2). For *evaluation and revision*, GLM-5.3-Flash serves as the rubric-scoring judge agent for model comparison, using source and runtime evidence. GLM-5.2 performs the diagnosis, design, and review roles that prepare revision instructions (Appendices A.3 and A.5). Candidate models execute tasks through Claude Code or OpenHands. For *implementation and writing*, Codex and Claude Code assist workflow development, and AI tools help draft and polish the manuscript. AI-assisted contributions undergo human review, and the authors take responsibility for the final content.

ETHICS STATEMENT

Trace2Eval draws on real histories of online coding-agent use. When using the service, users are informed that their data may be used for research after desensitization and provide explicit consent through signed agreements or informed-consent forms, conditional on protection of their personal privacy. The research data come from these consenting users and are desensitized before use. The public supplementary materials contain methodological documentation, rubric specifications, profile summaries, and selected case prompts; the full source-traffic collection is not included. These materials support the study of task completion and artifact quality and should not be used to identify or profile the original users. Reuse must respect confidentiality and applicable data and software licences. Human reference judgments support oversight of automated scoring (Appendix A.3), whose agreement varies across quality dimensions. The benchmark reflects a specific traffic population and expert-defined criteria, so its scores characterize artifact quality in that setting rather than universal user preferences or individual developer ability.

REPRODUCIBILITY STATEMENT

The accompanying anonymous repository provides workflow documentation, profile taxonomy and distribution tables, complete human and agent scoring anchors, and the original and feedback prompts for the revision case study. Section 3 defines the construction and scoring methods, with trace selection and context recovery detailed in Appendices A.1 and A.2. Appendix A.3 specifies dimension weights, human-reference calibration, repeated scoring, and aggregation, while Section 4 and Appendix A.4 document experimental configurations and component validation. Appendix A.5 describes revision inputs, preserved task states, task pairing, and uncertainty estimation, alongside analysis of recorded repairs; Appendix A.4.7 specifies the unguided comparison. These descriptions connect the workflow to its validation criteria. Machine-readable taxonomy and rubric files accompany the human-readable specifications in the repository, supporting inspection and reuse of the evaluation protocol.

REFERENCES

Anthropic. Claude 3.7 Sonnet and Claude Code. https://www.anthropic.com/news/claude-3-7-sonnet, 2025. Accessed: 2026-09-08.

Anthropic. Claude Fable 5 and Claude Mythos 5. https://www.anthropic.com/news/claude-fable-5-mythos-5, 2026. Accessed: 2026-09-08.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021. URL https://arxiv.org/abs/2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

Zhaoyang Chu, Jiarui Hu, Xingyu Jiang, Pengyu Zou, Han Li, Chao Peng, Peter O'Hearn, Earl T. Barr, Mark Harman, Federica Sarro, and He Ye. TerminalWorld: Benchmarking Agents on Real-World Terminal Tasks. *arXiv preprint arXiv:2605.22535*, 2026. URL https://arxiv.org/abs/2605.22535.

DeepSeek. DeepSeek-V4-Pro Model Card. https://huggingface.co/deepseek-ai/DeepSeek-V4-Pro, 2026. Accessed: 2026-09-26.

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.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, Yue Hou, Fei Hu, Zhaojian Li, Weiran Shi, Zaiyuan Wang, Daoguang Zan, Chenchen Zhang, Xiaoxu Zhang, Qizhi Chen, Xianfu Cheng, Bo Deng, Qingshui Gu, Kai Hua, Juntao Lin, Pai Liu, Mingchen Li, Xuanguang Pan, Zifan Peng, Yujia Qin, Yong Shan, Zhewen Tan, Weihao Xie, Zihan Wang, Yishuo Yuan, Jiayu Zhang, Enduo Zhao, Yunfei Zhao, He Zhu, Liya Zhu, Chenyang Zou, Ming Ding, Jianpeng Jiao, Jiaheng Liu, Minghao Liu, Qian Liu, Chongyang Tao, Jian Yang, Tong Yang, Zhaoxiang Zhang, Xinjie Chen, Wenhao Huang, and Ge Zhang. NL2Repo-Bench: Towards Long-Horizon Repository Generation Evaluation of Coding Agents. *arXiv preprint arXiv:2512.12730*, 2025. URL https://arxiv.org/abs/2512.12730.

Lingyue Fu, Xin Ding, Linyue Pan, Yaoming Zhu, Shao Zhang, Lin Qiu, Weiwen Liu, Weinan Zhang, Xuezhi Cao, Xunliang Cai, Jiaxin Ding, and Yong Yu. CATArena: Evaluating Evolutionary Capabilities of Code Agents via Iterative Tournaments. *arXiv preprint arXiv:2510.26852*, 2025. URL https://arxiv.org/abs/2510.26852.

Google. Gemini 3.6 Flash Model. https://ai.google.dev/gemini-api/docs/models/gemini-3.6-flash, 2026a. Accessed: 2026-09-26.

Google. Thought Signatures. https://ai.google.dev/gemini-api/docs/generate-content/thought-signatures, 2026b. Accessed: 2026-09-25.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence With APPS. *arXiv preprint arXiv:2105.09938*, 2021. URL https://arxiv.org/abs/2105.09938.

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. *arXiv preprint arXiv:2606.22883*, 2026. URL https://arxiv.org/abs/2606.22883.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974*, 2024. URL https://arxiv.org/abs/2403.07974.

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? *arXiv preprint arXiv:2310.06770*, 2023. URL https://arxiv.org/abs/2310.06770.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. DS-1000: A Natural and Reliable Benchmark for Data Science Code Generation. *arXiv preprint arXiv:2211.11501*, 2022. URL https://arxiv.org/abs/2211.11501.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Zhi Jin, Hao Zhu, Huanyu Liu, Kaibo Liu, Lecheng Wang, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yihong Dong, Yuqi Zhu, Bin Gu, and Mengfei Yang. DevEval: Evaluating Code Generation in Practical Software Projects. *arXiv preprint arXiv:2401.06401*, 2024. URL https://arxiv.org/abs/2401.06401.

Xuechen Li, Tianyi Zhang, Yann Dubois, Rohan Taori, Ishaan Gulrajani, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. AlpacaEval: An Automatic Evaluator of Instruction-following Models. https://github.com/tatsu-lab/alpaca_eval, 2023. Accessed: 2026-09-08.

Percy Liang, Rishi Bommasani, Tony Lee, Dimitris Tsipras, Dilara Soylu, Michihiro Yasunaga, Yian Zhang, Deepak Narayanan, Yuhuai Wu, Ananya Kumar, et al. Holistic Evaluation of Language Models. *arXiv preprint arXiv:2211.09110*, 2022. URL https://arxiv.org/abs/2211.09110.

Wenhan Liu, Jiajie Jin, Zhaoheng Huang, Tongyu Wen, Guanting Dong, Ziliang Zhao, Yutao Zhu, Zhicheng Dou, and Ji-Rong Wen. The Rules of the Game: A Survey of Rubrics for Large Language Models. https://openreview.net/pdf?id=FnSimngGYk, 2026. Accessed: 2026-09-08.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG Evaluation using GPT-4 with Better Human Alignment. *arXiv preprint arXiv:2303.16634*, 2023. URL https://arxiv.org/abs/2303.16634.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative Refinement with Self-Feedback. *arXiv preprint arXiv:2303.17651*, 2023. URL https://arxiv.org/abs/2303.17651.

Meituan. LongCat-2.0 Model Card. https://huggingface.co/meituan-longcat/LongCat-2.0, 2026. Accessed: 2026-09-26.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Jesse Hu, Christopher Michael Rytting, Ryan Marten, Yixin Wang, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces. *arXiv preprint arXiv:2601.11868*, 2026. URL https://arxiv.org/abs/2601.11868.

MiniMax. MiniMax-M3 Model Card. https://huggingface.co/MiniMaxAI/MiniMax-M3, 2026. Accessed: 2026-09-26.

Moonshot AI. Kimi-K3 Model Card. https://huggingface.co/moonshotai/Kimi-K3, 2026. Accessed: 2026-09-26.

OpenAI. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. Accessed: 2026-09-08.

OpenAI. Introducing Codex. https://openai.com/index/introducing-codex/, 2025. Accessed: 2026-09-08.

OpenAI. Why SWE-bench Verified no longer measures frontier coding capabilities. https://openai.com/index/why-we-no-longer-evaluate-swe-bench-verified/, 2026a. Accessed: 2026-09-08.

OpenAI. GPT-5.5 Model. https://developers.openai.com/api/docs/models/gpt-5.5, 2026b. Accessed: 2026-09-26.

OpenAI. GPT-5.6 sol Model. https://developers.openai.com/api/docs/models/gpt-5.6-sol, 2026c. Accessed: 2026-09-08.

Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B. Hashimoto. Proving Test Set Contamination in Black Box Language Models. *arXiv preprint arXiv:2310.17623*, 2023. URL https://arxiv.org/abs/2310.17623.

Hao Wang, Hanchen Li, Qiuyang Mang, Alvin Cheung, Koushik Sen, and Dawn Song. Do Androids Dream of Breaking the Game? Systematically Auditing AI Agent Benchmarks with BenchJack. *arXiv preprint arXiv:2605.12673*, 2026a. URL https://arxiv.org/abs/2605.12673.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. SWE-Mirror: Scaling Issue-Resolving Datasets by Mirroring Issues Across Repositories. *arXiv preprint arXiv:2509.08724*, 2025. URL https://arxiv.org/abs/2509.08724.

Xin Wang, Liangtai Sun, Yaoming Zhu, Shuang Zhou, Jiaxing Liu, Fengjiao Chen, Lin Qiu, Xuezhi Cao, Xunliang Cai, Licheng Zhang, and Zhendong Mao. Asuka-Bench: Benchmarking Code Agents on Underspecified User Intent and Multi-Round Refinement. *arXiv preprint arXiv:2606.05920*, 2026b. URL https://arxiv.org/abs/2606.05920.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, and Chenghua Lin. Large-Scale Terminal Agentic Trajectory Generation from Dockerized Environments. *arXiv preprint arXiv:2602.01244*, 2026. URL https://arxiv.org/abs/2602.01244.

John Yang, Kilian Lieret, 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. *arXiv preprint arXiv:2504.21798*, 2025. URL https://arxiv.org/abs/2504.21798.

Z.ai. GLM-5.2 Model Card. https://huggingface.co/zai-org/GLM-5.2, 2026a. Accessed: 2026-09-26.

Z.ai. GLM-5.3 Model Card. https://huggingface.co/zai-org/GLM-5.3, 2026b. Accessed: 2026-09-26.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. *arXiv preprint arXiv:2306.05685*, 2023. URL https://arxiv.org/abs/2306.05685.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. TermiGen: High-Fidelity Environment and Robust Trajectory Synthesis for Terminal Agents. *arXiv preprint arXiv:2602.07274*, 2026. URL https://arxiv.org/abs/2602.07274.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *arXiv preprint arXiv:2406.15877*, 2024. URL https://arxiv.org/abs/2406.15877.

A SUPPLEMENTARY METHODS AND RESULTS

The appendix expands the construction and evaluation protocols in the order used by the main text. The final benchmark contains 129 tasks retained from 300 algorithm-selected candidates through automated screening and targeted human review. Component-validation samples and operational score summaries retain their measured denominators.

Appendix A.1 expands the data construction in §3.1, covering reporting populations, taxonomy, selection, and distribution bias. Appendix A.2 details the recovery and replay protocol in §3.2, while Appendix A.3 provides the rubric and scoring specifications underlying §3.3.

Appendix A.4 supplies the extended validation and model results for §4, and Appendix A.5 describes the revision protocol and initial paired results in §4.4. Appendix A.5.3 follows one request through feedback and repair. Appendix A.6 summarizes the profile taxonomy; complete specifications and pipeline inputs and outputs are available in the anonymous repository.

A.1 DATA CONSTRUCTION AND DISTRIBUTION

A.1.1 TRACE SELECTION AND BENCHMARK CONSTRUCTION

The construction process in §3.1 turns online interactions into independently evaluable coding tasks. Claude Sonnet 4.6 performs trace labeling and automated admission judgments. Coding-request identification reduces approximately 20,000 collected sessions to 14,603 code-related traces, from which distribution-aligned greedy selection draws 300 candidates. A second greedy stage combines multi-round voting under the admission rubric with the same distributional objective to select the final 129 tasks; human reviewers resolve borderline admission judgments. Table 3 summarizes these stages and their construction decisions.

Table 3: Trace counts and selection criteria at each benchmark-construction stage.

Stage	Size	Construction step
Collected sessions	$\approx$ 20,000	Desensitized online coding-agent traffic.
Coding population $\mathbb{C}$	14,603	Identify coding requests and assign profile labels.
Selected candidates $\mathbb{S}$	300	Apply distribution-aligned greedy selection to the coding population.
Final benchmark $\mathbb{B}$	129	Combine rubric-based voting and distribution constraints in a second greedy stage; review borderline cases.

Identify coding requests and summarize user queries. We begin with 20,000 desensitized sessions collected from online coding-agent traffic between June and July 2026 by randomly sampling approximately equal numbers per day. We label these real user traces for coding relevance and summarize their user queries. Requests involving code development, modification, or analysis form the coding population $\mathbb{C}$, containing 14,603 traces. Each query summary captures the user's objective and explicit requirements from the interaction, providing a concise description for subsequent profile labeling and candidate selection.

Characterize the coding population with user profiles. The traces in $\mathbb{C}$ receive complementary labels describing the intended activity, technical context, application setting, and requester role. Task-type axes `x1`/`x2` and technical-domain axes `z1`/`z2` distinguish broad families from finer activities and specializations. Usage scenario `y` and user role `w` distinguish requests with similar technical content but different purposes or user groups. This labeling step supplies the population proportions used for selection, preserving distinctions that a single coding-related label would miss. Using the taxonomy summarized in Appendix A.6, we label and select online traces and SWE-bench instances, finding that SWE-bench does not match the online traffic distribution (Appendix A.1.4).

Select a representative candidate set. Candidate selection and admission screening aim to assemble tasks with sufficient difficulty to reveal performance differences across agents while remaining aligned with real user-profile distributions. Algorithm 1 addresses the distributional objective by selecting $\mathbb{S} \subseteq \mathbb{C}$ with $|\mathbb{S}| = 300$. Category quotas reflect the population proportions, and greedy

selection jointly fills deficits across profile axes. The second greedy stage applies the admission rubric while retaining the distributional objective, jointly considering task difficulty, discriminative value, and profile alignment. Appendix A.1.3 details the selection procedure, and Appendix A.1.4 examines the profile distributions of the 300 selected traces.

Define the admission rubric for candidate screening. We develop the human-defined admission rubric through experiments on a small calibration set. Table 4 summarizes the criteria used to screen the 300 selected candidates $\mathbb{S}$ into the final benchmark $\mathbb{B}$. The criteria favor substantive implementation or code-reasoning demands expected to distinguish agent capabilities, while checking task clarity, recoverability, and distinctiveness. Together with distribution-aligned selection, this screening aims to provide informative comparisons across the range of requests represented in real coding traffic. The artifact-quality rubric in Appendix A.3.1 is used later to grade the agent's delivered result.

Table 4: Admission rubric used to screen selected candidates through automated voting and targeted human review.

Criterion	Admission decision rules
Task clarity	**Retain:** The objective, constraints, and task boundary can be recovered from the interaction.
	Exclude: Fragments or continuation requests whose essential context cannot be recovered.
Task difficulty	**Retain:** Substantive implementation or code reasoning is expected to reveal differences in agent performance.
	Exclude: Trivial edits, pure information lookup, or setup-only requests without substantive coding work.
Input recoverability	**Retain:** Essential files and data are observable or can be reconstructed from available evidence.
	Exclude: Missing private inputs or insufficient evidence prevent constructing the required starting context.
Environment feasibility	**Retain:** Required execution can be represented in the supported environment.
	Exclude: Essential authenticated services, remote systems, or internal platforms cannot be reproduced.
Task distinctiveness	**Retain:** The request contributes a distinct task objective or is the retained representative of a duplicate group.
	Exclude: Another retained request already covers an equivalent core objective and expected result.

Combine admission voting with distribution-constrained greedy selection. Across screening rounds, each candidate in $\mathbb{S}$ receives five independent automated keep-or-remove judgments under the admission rubric. At least three agreeing votes determine the initial keep-or-remove decision, while cases near the voting threshold are referred for human reinspection. Reviewers examine these cases against the same criteria and resolve the admission decision, focusing human effort on ambiguous requests without requiring a full manual pass over all 300 candidates. Semantic deduplication retains representatives of equivalent task objectives. The second greedy stage combines these admission decisions with population-profile constraints to retain $\mathbb{B} \subseteq \mathbb{S}$ with $|\mathbb{B}| = 129$, preserving distribution alignment while satisfying the admission criteria.

Use trace labels to guide downstream recovery. Alongside user-profile categories, trace labeling provides initial judgments of whether a request is code-related and whether it requires workspace or environment recovery. These judgments draw on information in the original interaction and give downstream agents a preliminary reference for identifying the task's file and runtime requirements. The retained traces, query summaries, and labels are then passed to the recovery workflows in Appendix A.2. As illustrated in Figure 9, specialized agents use this information to recover the

required files and reconstruct the execution environment, followed by trajectory replay to validate the recovered context.

A.1.2 Profile Taxonomy and Reference Distributions

The anonymous repository provides the 171-category dictionary used in §3.1 and the full traffic and SWE-bench distribution tables. The traffic table concerns all 20,000 collected sessions, not only the 14,603 coding sessions. The SWE-bench comparison pools 1,531 instances from the three test splits listed in Table 5, each characterized under the same profile taxonomy. Category labels are translated for presentation without changing their identities.

The task-type hierarchy describes what users ask an agent to do. Axis `x1` groups broad activities such as task execution, programming, and information retrieval, while `x2` distinguishes operations such as feature implementation, bug fixing, code understanding, and automation orchestration. These predefined categories provide a consistent vocabulary for recurring activities across domains. Online requests, however, can combine several objectives or involve specialized workflows that do not map cleanly to a single activity category, even when their application setting or technical domain is identifiable. Allowing an unassigned task type accommodates this open-ended scope without forcing a category assignment. N/A records the absence of an assigned task-type label; the remaining profile axes can still characterize the request.

In the full traffic table, task-type labels are unassigned for 10,354 sessions (51.77%) on each of `x1` and `x2`. We omit these N/A entries and normalize each task-type distribution over its 9,646 labeled sessions, so the reported proportions describe labeled tasks rather than all traffic. The other four axes retain N/A and use all 20,000 sessions as their denominator. Counts remain observed category frequencies. The coding-only comparisons in Figures 2, 7, and 8 use the same label export restricted to code-related sessions; their task hierarchy likewise excludes unassigned labels and uses its own labeled-sample denominator.

We distinguish presence coverage from frequency alignment in Appendix A.1.4. Categories aggregated as Others in a compact figure remain individually represented in the repository tables.

A.1.3 Distribution-Aligned Candidate Selection

The selection step in Appendix A.1 selects 300 candidates from the 14,603 code-related traces for subsequent screening under Table 4. Each coding trace carries labels for task type, usage scenario, technical domain, and user role, so selecting it changes several profile distributions at once. Algorithm 1 coordinates these choices through category quotas and iterative deficit reduction. This is the first of two greedy stages; the second combines rubric-based admission with distribution constraints to select the final benchmark.

Translate profile proportions into target counts. Let $\mathbb{C}$ denote the 14,603 code-related traces, $\mathbb{S} \subseteq \mathbb{C}$ the selected candidate set, and $n_{\text{sel}} = 300$ the selection budget. Each trace $\tau \in \mathbb{C}$ has a label $\ell_a(\tau)$ on each profile axis $a \in \{\texttt{x1}, \texttt{x2}, \texttt{y}, \texttt{z1}, \texttt{z2}, \texttt{w}\}$. For category v on axis a, $p_{a,v}$ is its empirical proportion in $\mathbb{C}$, and $c_{a,v}$ counts its occurrences in the current set $\mathbb{S}$. The fractional allocation $q_{a,v} = n_{\text{sel}} p_{a,v}$ gives the desired number of selected traces carrying that label. We obtain integer targets $t_{a,v}$ by taking the floor of each allocation and assigning the remaining seats to the largest fractional remainders. The targets on each axis sum to n_{sel}. Because each trace contributes labels on several axes, these targets guide a joint approximation to the population distributions.

Select traces that fill several deficits. At each iteration, we score every unselected trace by the remaining target fractions of its labels,

$$\text{Deficit}(\tau) = \sum_a \frac{\big(t_{a,\ell_a(\tau)} - c_{a,\ell_a(\tau)}\big)^+}{\max\big(t_{a,\ell_a(\tau)}, 1\big)}, \qquad \tau \in \mathbb{C} \setminus \mathbb{S}, \tag{5}$$

where $(x)^+ = \max(x, 0)$. A category contributes its unfilled target fraction until its count reaches the target, after which its contribution is zero. Normalization makes relative shortfalls comparable across common and rare categories, and the denominator remains well defined when rounding gives a zero target. A trace can therefore address several shortfalls at once, such as an underrepresented

task type and usage scenario. Algorithm 1 selects the trace with the largest score, updates the counts of all its labels, and recomputes priorities for the next iteration. This procedure builds a set of 300 candidates while balancing the six profile axes without separately sampling every combination of labels.

Algorithm 1 Distribution-aligned candidate selection.

Require: labeled coding population $\mathbb{C}$ with labels $\ell_a(\tau)$; population proportions $p_{a,v}$; budget $n_{\text{sel}} = 300$
Ensure: candidate set $\mathbb{S}$ and its per-axis category proportions

for each axis a **do**
  $q_{a,v} \leftarrow n_{\text{sel}} p_{a,v}$ and $t_{a,v} \leftarrow \lfloor q_{a,v} \rfloor$ for each category v
  Assign the remaining $n_{\text{sel}} - \sum_v t_{a,v}$ seats by descending fractional remainder
end for
$\mathbb{S} \leftarrow \emptyset$; $c_{a,v} \leftarrow 0$ for every axis a and category v
while $|\mathbb{S}| < n_{\text{sel}}$ and $\mathbb{C} \setminus \mathbb{S} \neq \emptyset$ **do**
  Compute $\text{Deficit}(\tau)$ for each $\tau \in \mathbb{C} \setminus \mathbb{S}$ using Equation (5)
  $\tau^\star \leftarrow \arg\max_{\tau \in \mathbb{C} \setminus \mathbb{S}} \text{Deficit}(\tau)$, breaking ties by a fixed trace ordering
  $\mathbb{S} \leftarrow \mathbb{S} \cup \{\tau^\star\}$
  $c_{a,\ell_a(\tau^\star)} \leftarrow c_{a,\ell_a(\tau^\star)} + 1$ for each axis a
end while
Return $\mathbb{S}$ and proportions $c_{a,v}/|\mathbb{S}|$ for every axis a and category v

Retain distribution constraints during admission. The algorithm returns the 300 selected candidates $\mathbb{S}$, which then enter the admission workflow described in Appendix A.1. Using the rubric in Table 4, five independent automated judgments assess each candidate's suitability for informative agent comparisons, and human reviewers resolve borderline cases. A second greedy stage combines multi-round admission decisions with the same population-profile objective to select the 129-task benchmark $\mathbb{B} \subseteq \mathbb{S}$. Both stages therefore preserve distribution alignment while the second also enforces task difficulty and feasibility. To evaluate distribution alignment, we compare the candidates and SWE-bench with the same coding population under the same taxonomy; Appendix A.1.4 presents the distributions in Figures 2, 7, and 8 and their distances in Table 6.

A.1.4 User-Profile Coverage and Distribution of the Candidate Set

We evaluate the distribution-matching step in Appendices A.1 and A.1.3 using the full coding population $\mathbb{C}$ of 14,603 traces. Applying Algorithm 1 at a budget of 300 produces the candidate set $\mathbb{S}$ for this analysis. We compare both $\mathbb{S}$ and a pooled SWE-bench comparison set $\mathbb{W}$ against the same population $\mathbb{C}$ under the taxonomy in Appendix A.1.2. Here, $\mathbb{W}$ pools the test instances from SWE-bench Verified, Multilingual, and Pro. Together, they contribute 1,531 instances, as detailed in Table 5. Each SWE-bench instance contributes once to the pooled distribution, with its problem statement labeled using the same six profile axes as online requests. Figures 2, 7, and 8 compare the resulting task, context, and technical-domain distributions. These visual comparisons characterize coverage of common requests and differences in their relative frequencies; the candidate set spans a broader range of observed activities than the SWE-bench comparison set.

Table 5: SWE-bench test sets pooled for the user-profile comparison.

Source benchmark	Split	Instances
SWE-bench Verified	Test	500
SWE-bench Multilingual	Test	300
SWE-bench Pro	Public test	731
Combined comparison set $\mathbb{W}$		1,531

Measuring distribution alignment. We denote the individual benchmarks by $\mathbb{W}_{\text{Verified}}$, $\mathbb{W}_{\text{Multi}}$, and $\mathbb{W}_{\text{Pro}}$, and retain $\mathbb{W}$ for their pooled set. Figures 2, 7, and 8 show the pooled SWE-bench distribution as an overview, while Table 6 additionally compares each benchmark separately. For

any comparison set $\mathbb{T}$, let $n^{\mathbb{T}}_{a,v}$ count traces with category v on axis a, and let $p^{\mathbb{T}}_{a,v} = n^{\mathbb{T}}_{a,v}/|\mathbb{T}|$ be the corresponding proportion. With $p^{\mathbb{C}}_{a,v}$ defined analogously for the coding population, total variation measures their distributional distance,

$$\mathrm{TV}_a(\mathbb{C}, \mathbb{T}) = \frac{1}{2} \sum_v \left| p^{\mathbb{T}}_{a,v} - p^{\mathbb{C}}_{a,v} \right| . \tag{6}$$

The sum covers the same categories for both sets, assigning zero to categories absent from a set. The factor of one half avoids counting both the excess and the shortfall of the same probability mass. TV ranges from zero for identical distributions to one for disjoint distributions; a value of 0.02 means that 2% of the probability mass would need to move between categories to match the reference. Table 6 uses the full per-axis category counts and sample-set denominators, before any grouping used for visualization. This common reference makes TV a direct test of whether each set represents user activities at the frequencies observed in real traffic.

Covering task activities and user contexts. Figure 2 shows why repository benchmarks offer a selective view of online demand. Among traces with assigned task types, programming accounts for 100.0% of the SWE-bench comparison set, versus 53.1% of coding traffic and 53.2% of selected candidates. Bug fixing alone contributes 1,032 of 1,531 SWE-bench instances (67.4%), compared with 1,046 of 7,590 labeled coding traces (13.8%) and 22 of 158 labeled candidates (13.9%). The selected set includes both programming and other common activities. Coverage also extends across user contexts. Programming scenarios account for 98.6% of SWE-bench, but 41.8% of coding traffic and 42.0% of candidates. Likewise, software developers constitute 81.5% of SWE-bench profiles, compared with 68.0% in both the coding population and the selected set. SWE-bench therefore provides a narrower view of the observed mix of tasks, application contexts, and user roles. The candidate set covers a broader range of common requests.

Figure 6 summarizes the technical-domain hierarchy.

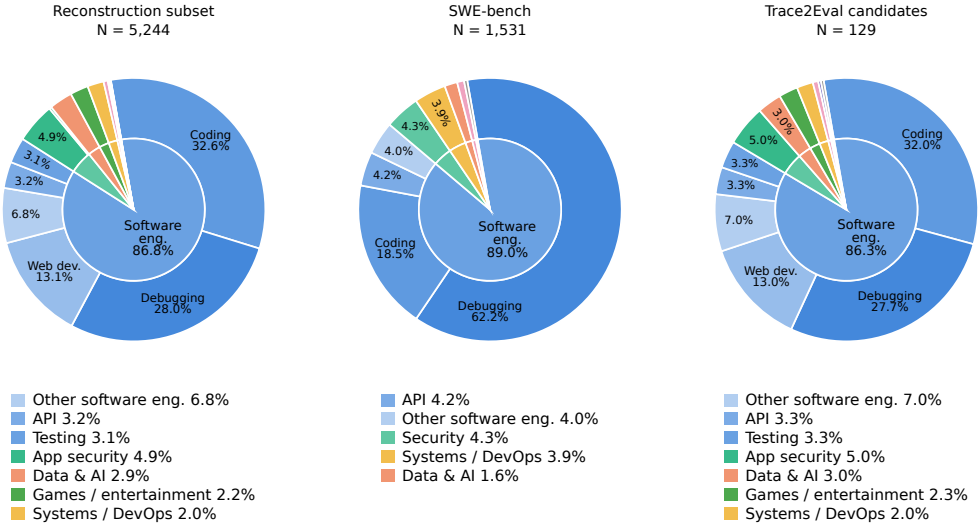


Figure 6: Technical-domain profiles of the reconstruction subset, SWE-bench, and selected candidates. Inner disks show broad domains, and outer rings show activities within them.

Preserving broad domains and specific activities. Figure 8 shows that SWE-bench is misaligned with online demand at both domain and activity levels, overrepresenting software engineering and debugging while underrepresenting web development. Debugging accounts for 62.2% of SWE-bench, compared with 14.2% of coding traffic and 14.3% of Trace2Eval candidates. The candidates include these common domains and activities, extending coverage beyond the debugging-heavy profile of SWE-bench.

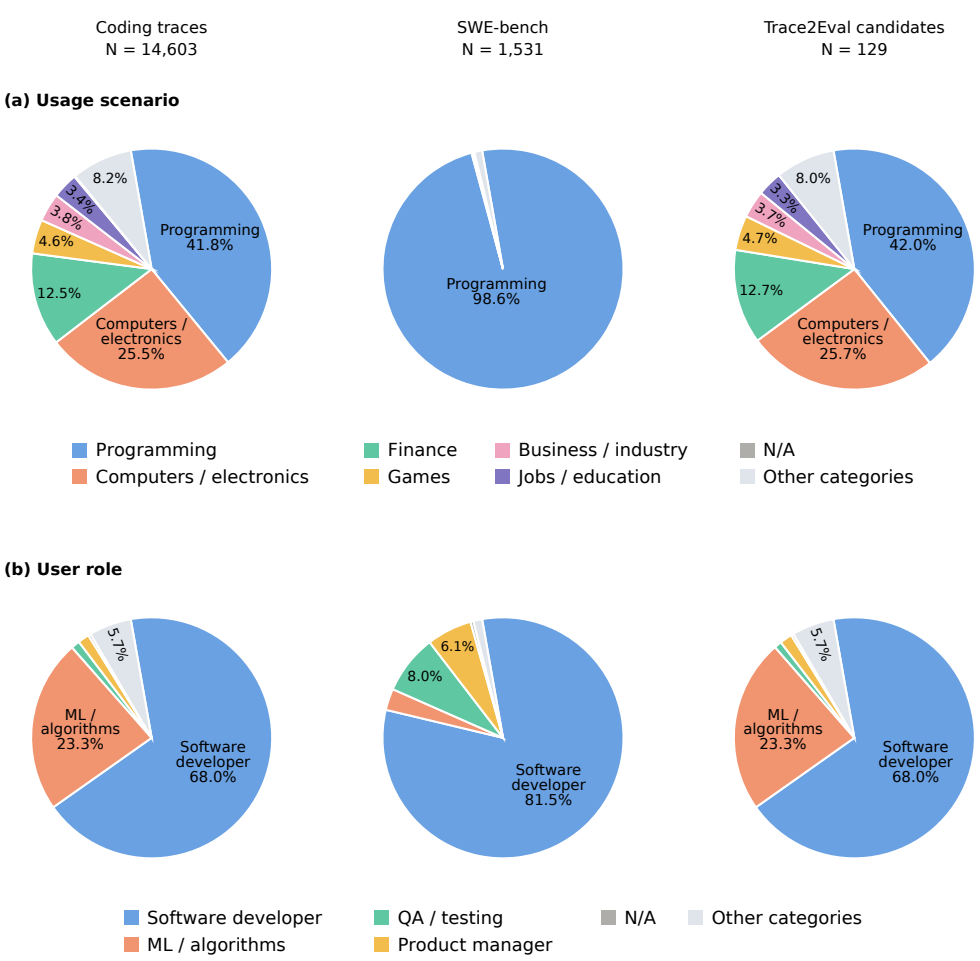


Figure 7: Usage scenarios (a) and user roles (b) in coding traces, SWE-bench, and selected candidates. The same colors identify corresponding categories across the three populations.

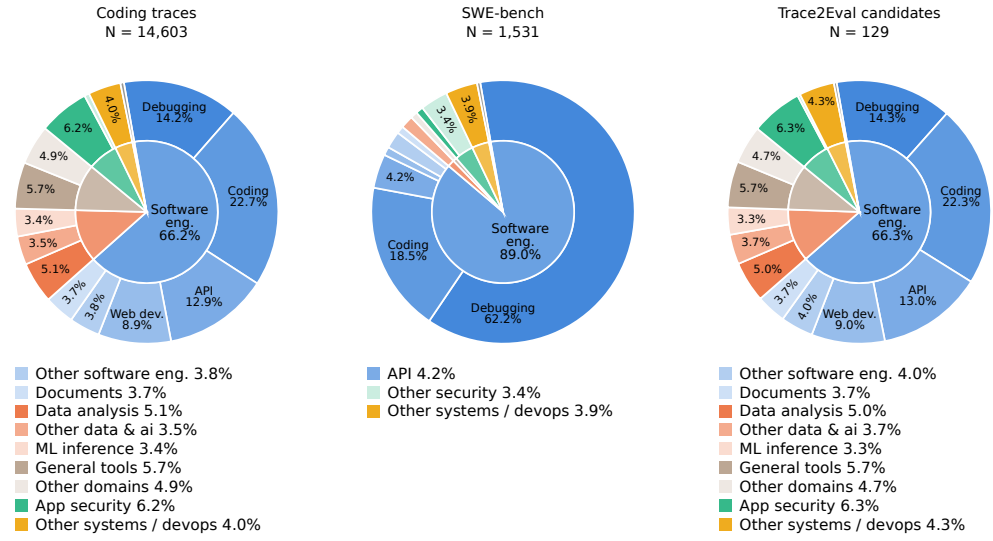


Figure 8: Technical-domain profiles of coding traces, SWE-bench, and selected candidates. Inner disks show broad domains, and outer rings show activities within them.

Table 6: Total variation (TV) between each comparison set and the coding population $\mathbb{C}$ across six profile axes. Lower values indicate closer alignment. $\mathbb{W}$ combines the three SWE-bench variants; $\mathbb{S}$ denotes the selected Trace2Eval candidates.

	$\mathrm{TV}_a(\mathbb{C}, \mathbb{T}) \downarrow$				
Profile axis	$\mathbb{W}_{\text{Verified}}$	$\mathbb{W}_{\text{Multi}}$	$\mathbb{W}_{\text{Pro}}$	$\mathbb{W}$ (total)	$\mathbb{S}$
`x1` - Task type L1	0.7241	0.7241	0.7241	0.7241	**0.0120**
`x2` - Task type L2	0.8453	0.8114	0.7887	0.7877	**0.0191**
`y` - Usage scenario	0.5795	0.5815	0.5664	0.5721	**0.0220**
`z1` - Technical domain L1	0.2736	0.2846	0.2252	0.2287	**0.0068**
`z2` - Technical domain L2	0.6932	0.7341	0.4551	0.5318	**0.0232**
`w` - User role	0.2224	0.2784	0.2733	0.2492	**0.0060**

Alignment across all six profile axes. Table 6 quantifies the contrast in Figures 2, 7, and 8: the 300-candidate set has lower measured distributional distance from the coding population than SWE-bench on all six axes. Candidate TV ranges from 0.0060 to 0.0232, compared with 0.2287 to 0.7877 for pooled SWE-bench. The candidates also have lower TV than every individual SWE-bench variant on every axis. These distances describe the algorithm-selected candidate set before admission screening.

The observed user profiles guide coverage of common requests during candidate selection. The second greedy stage combines multi-round rubric voting with distribution constraints to produce the 129-task benchmark, jointly preserving profile alignment and evaluation value. The resulting benchmark is intended to cover diverse real-world coding needs; the artifact rubrics in Appendix A.3.1 assess delivery quality within those task contexts.

A.2 Workspace and Environment Recovery

Recovery turns a retained user request into an executable task by reconstructing its initial files and dependencies where needed. We use Claude Opus 4.6 for the file and environment recovery workflows. An agent first assesses recovery needs from the previously summarized query and supporting evidence it locates in the trace, assigning each task to no recovery, file recovery only, or both file and environment recovery. When both are needed, files are recovered first so that environment inference uses the restored workspace. Following §3.2, q denotes the request and $\tau = (T_1, \ldots, T_T)$ its tool trajectory, with $T_t = (u_t, i_t, o_t)$ recording the tool, input, and observed output. The targets W_0 and E_0 are the workspace and environment before the recorded solution begins; hats denote recovered estimates. Figure 9 shows how file recovery supplies $\widehat{W}_0$, environment inference supplies $\widehat{E}_0$, and replay checks their joint support for the recorded actions.

A.2.1 File Recovery

Restore observed inputs and identify missing contents. The restorer agent first extracts pre-existing files from tool reads, user-provided text, and embedded context. For files modified during the session, it preserves the observed earlier contents; files created or downloaded as part of the recorded solution are excluded from the initial workspace. A shared path mapping preserves directory relationships when moving inputs into the recovered workspace. This gives subsequent agents a common reference for connecting file evidence, imports, and tool arguments.

The resulting inventory distinguishes fully observed inputs, partially observed inputs, missing task-relevant contents, potential public resources, and task-irrelevant placeholders. It also identifies supporting files implied by imports or configuration references. Fully observed files require no generation. Missing contents are assigned either to contextual completion or to public retrieval, while unrelated files can remain empty placeholders. This division directs reconstruction effort toward inputs that affect the user's task.

Complete missing contents with a mocker agent. A mocker agent operates on one assigned file, using the user request q, its recorded tool trajectory τ, and the restored files as context. For a partially observed file, it fills the missing portions while preserving the visible contents. For a missing task-relevant file, it reconstructs a compatible implementation or input from referenced interfaces,

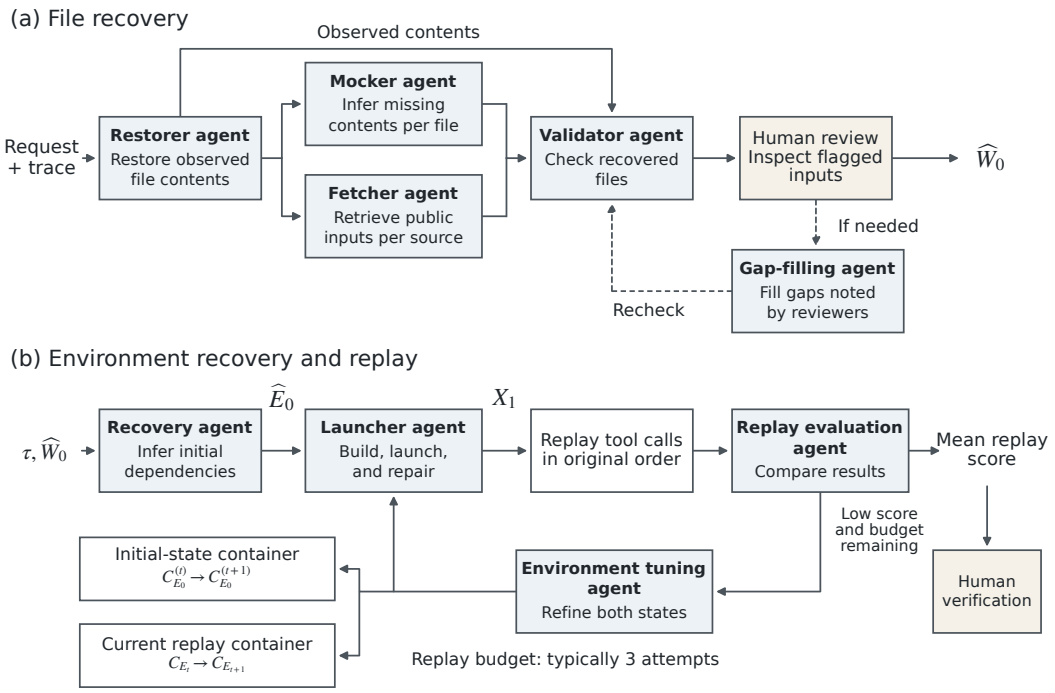


Figure 9: Workspace and environment recovery. (a) Recovery agents restore observed file contents, infer missing contents, retrieve public inputs, and validate the assembled workspace, with human review guiding further corrections. (b) Tool calls are replayed in order, and their results guide updates to the initial and current environment states. Replay is limited to a preset budget, typically three attempts, before human verification. Here τ denotes the recorded trajectory and $X_1 = (\widehat{W}_0, \widehat{E}_0)$ the recovered starting context.

fields, imports, and usage. Public templates can inform standard configuration or library usage, whereas project-specific contents are inferred from the surrounding evidence. Other files remain read-only so that parallel completion does not overwrite the shared evidence.

More formally, let f_{obs} be the observed initial contents of a file at positions $\mathbb{O}_f$, and let $\widehat{f}$ be its completed version. The preservation constraint is

$$\widehat{f}\big|_{\mathbb{O}_f} = f_{\text{obs}}, \tag{7}$$

where restriction to $\mathbb{O}_f$ selects the observed portions. Complete observations fix the entire file; partial observations constrain only the visible portions. The remaining contents must fit the request and neighboring interfaces, and are recorded as inferred rather than directly recovered.

Retrieve public datasets and resources. The fetcher agent handles inputs such as public datasets, model weights, and resource bundles whose authentic contents can be obtained externally. Unlike the mocker agent, which is dispatched per file, the fetcher agent works on files grouped by source so that related inputs share the same dataset or resource version. It first verifies the source using names, versions, and usage in the trace, then retrieves the required files and checks their placement, format, and whether the downloaded files are nonempty.

Source verification also determines whether retrieval is appropriate. A user-modified file or an unsupported public-source attribution is reassigned to contextual completion when its contents can be inferred. Failed downloads and unavailable task-critical private inputs remain explicit gaps. Thus, public retrieval preserves authentic data where available, while outputs from the mocker agent supply inferred context without being mistaken for original dataset contents. Independent file completions and source-group retrievals can proceed in parallel before their outputs are merged.

Validate the assembled workspace and resolve review findings. The validator agent checks the recovered inventory against the files on disk and inspects key files for compatible imports, exports, configuration references, and meaningful contents. Evidence records connect restored portions to their trace observations and distinguish inferred contents from retrieved resources. These records support targeted inspection of uncertain inputs. When human review identifies a missing input, a separate gap-filling agent uses the reviewer notes and original trace to supplement the existing workspace, followed by checks of the requested additions and syntax. The resulting $\widehat{W}_0$ is the fixed starting snapshot for environment recovery and replay. The acceptance study checks recovered files against the request and trace evidence, recording missing inputs and unintended solution artifacts; Table 10 in Appendix A.4.1 gives the results.

A.2.2 Environment Recovery and Replay

Infer the initial environment through temporal evidence. The recovery agent combines $\widehat{W}_0$ with installation commands, runtime indicators, errors, service usage, and configuration changes in τ. A feasibility check first identifies which platform dependencies can be represented in the execution environment. The update in Eq. 1 maintains two container states throughout replay. C_{E_t} is the current environment before call t, and $C_{E_0}^{(t)}$ is the initial-environment estimate after the preceding calls in that round. The current container begins each round from the latest initial-state estimate. Processing T_t advances the current environment and provides evidence for correcting both containers: missing initial dependencies require changes to the initial-state container as well as the current replay state. The index t follows tool calls within a round; subsequent rounds repeat this process with the revised initial environment.

Successful use of a dependency before any installation supports its inclusion in $\widehat{E}_0$. An explicit later installation instead belongs to the inferred change $\widehat{\Delta E}_t$ at that step. For example, a package first installed midway through the trace should become available at that point during replay, rather than being preinstalled in the initial image. The initial-state container $C_{E_0}^{(T+1)}$ at the end of a round realizes $\widehat{E}_0$, while inferred changes between successive current states define $\widehat{\Delta E}_t$. Thus, $\widehat{E} = (\widehat{E}_0, \widehat{\Delta E}_1, \ldots, \widehat{\Delta E}_T)$ describes the recovered starting environment and its subsequent evolution. Initial requirements are mapped to a supported Linux configuration and packaged into a container. A launcher agent builds and cold-starts it, repairs configuration issues revealed by build or startup failures, and checks that it is ready for replay.

Replay the trajectory from the recovered context. Since a complete ground-truth workspace and environment are unavailable, we assess recovery through trajectory equivalence: agreement between replayed tool-call results and their recorded observations. First, we extract an ordered list of tool calls from τ, retaining each tool u_t, input i_t, and original observation o_t. Second, we cold-start the recovered environment $\widehat{E}_0$ with a working copy of $\widehat{W}_0$ and replay the calls in their original order. Third, we compare each eligible replay observation a_t with its recorded counterpart o_t and aggregate the per-call agreement scores. These comparisons test recovery fidelity and locate discrepancies for subsequent refinement.

Within a replay round, let X_t denote the joint workspace and current environment state immediately before call t, with $X_1 = (\widehat{W}_0, \widehat{E}_0)$. Its environment component is C_{E_t}; the initial-state container $C_{E_0}^{(t)}$ is maintained separately. For an executed action,

$$(X_{t+1}, a_t) = \mathrm{Exec}\big(X_t, u_t, \rho(i_t)\big), \tag{8}$$

where ρ applies the shared path mapping to tool inputs and a_t is the actual observation. Executed writes, installations, and service operations update the replay state in trajectory order. The changes $\widehat{\Delta E}_t$ explain these transitions and are not additional actions applied a second time. Replay modifies a working copy of the workspace, preserving the recovered file snapshot while allowing both environment containers to be refined.

Compute the replay score for recovery acceptance. For each replayed tool call, a replay evaluation agent compares the result a_t produced in the recovered workspace and environment with the original result o_t, using the tool u_t and its input i_t as context. It assigns an agreement score $m_t \in [0, 1]$:

identical results receive 1, while increasingly different results receive lower scores, approaching 0 for substantially different outcomes. These judgments form the trajectory-equivalence rubric; their arithmetic mean is the *replay score* in Eq. 3. A score of 1 after replaying and scoring the complete tool-call sequence indicates an equivalent working context for the recorded trajectory. We use this score to assess overall recovery quality for acceptance, while low-scoring calls locate discrepancies for refinement.

Use replay discrepancies to refine the recovered context. The automated refinement loop has a preset replay budget, typically three attempts. In each round, low-scoring tool calls identify file or environment discrepancies for targeted repair. Recovery agents correct the affected workspace files, while the environment tuning agent updates both the initial-state and current-state containers. When discrepancies remain and the budget permits, the launcher agent rebuilds and cold-starts the revised initial environment with the updated recovered workspace for another round. Each round compares tool-call results against the same recorded observations to verify these corrections. The resulting context and replay evidence then undergo human verification for recovery acceptance, as shown in Figure 1. The recovery studies assess acceptance (Table 11), compare single-shot recovery, launch repair, and full recovery (Table 12), and measure replay scores before and after refinement on the same cases; Appendix A.4.2 analyzes the outcomes.

A.3 Rubric and Scoring Protocol

A.3.1 Rubric Design and Human Annotation

Open-ended coding requests admit multiple valid implementations, so artifact quality must be judged against the user's needs rather than a unique reference solution. The rubric in §3.3 separates accuracy, code quality, and user experience into eleven dimensions. Its construction follows two stages on calibration artifacts. We first refine the human criteria to improve agreement between annotators and establish a stable reference, then refine the agent's scoring points to improve agreement with that fixed human reference. Ground truth (GT) denotes the final human reference from a third annotation round that integrates the first two annotators' scores and rationales with direct artifact inspection.

Ground the dimensions in human judgments. The dimensions are informed by a qualitative analysis of human annotations and the written reasons for preferring one coding output over another. Table 7 summarizes the concerns expressed in those rationales. A concern's frequency is the proportion of reviewed annotation results whose rationale mentions that concern, using the same collection of annotations as the denominator for every row. It measures how often a consideration enters a human judgment.

A rationale can cite several concerns, such as incomplete functionality together with a high repair cost, so the categories overlap and their proportions need not sum to 100%. Related concerns can also motivate the same rubric dimension. Functional completeness appears in 58.5% of the reviewed rationales and executability in 33.8%, supporting separate checks of whether the requested functionality is implemented and whether the artifact runs. Repair cost appears less frequently, but captures a different practical concern about how much work remains for the user. These frequencies inform the choice and scope of dimensions; they are not used as scoring weights.

Starting from these concerns, we collect expert judgments of how strongly each quality dimension affects user experience in real development. Experts assess the consequences of defects for task completion, usability, and subsequent maintenance or repair. These judgments establish group weights of 50% for accuracy, 35% for code quality, and 15% for user experience, as well as the within-group priorities in Table 8. Subsequent reassessment confirms the final weighting scheme used in the reported model scores. These weights preserve expert priorities in artifact-quality aggregation and human-reference agreement summaries and remain fixed during agent calibration.

Fix expert weights and evidence requirements. Let c index the three quality groups, λ_c its group weight, and $\mathbb{D}_c$ its dimensions. Each dimension d has an expert-assigned relative priority r_d within its group. We convert these priorities into the final dimension weights by normalization:

$$w_d = \lambda_c \frac{r_d}{\sum_{d' \in \mathbb{D}_c} r_{d'}}, \qquad d \in \mathbb{D}_c, \qquad \sum_d w_d = 1. \tag{9}$$

Table 7: User-quality concerns identified in human annotation rationales and their corresponding rubric dimensions. Frequencies indicate how often each concern is mentioned; a rationale may include several concerns.

Human concern	Rubric dimension	Frequency (%)
Requested functionality	Requirement implementation	58.5
Executability and runtime setup	Code execution	33.8
Trustworthy output data	Data authenticity	10.8
Robustness and error handling	Robustness	26.2
Architecture and technology fit	Architecture and technical choice	16.9
Clarity of the solution and its logic	Readability; requirement implementation	12.3
Runtime and memory efficiency	Time/space complexity	6.2
Effort and risk of repair	Repair cost	4.6
Layout and interaction usability	Interface and interaction	13.8
Useful additional functionality	Feature richness	10.8
Organization of displayed information	Information presentation	6.2
Implementation-constraint compliance	Requirement implementation	1.5
Redundant or missing explanations	Readability; information presentation	3.1

Table 8: Evidence requirements and final expert-informed weights for artifact scores and human-reference agreement summaries. Within each group, priorities r_d are normalized and multiplied by the group weight λ_c to obtain w_d.

Dimension	Evidence	Priority r_d	Weight w_d (%)
Accuracy $\lambda_c = 50\%$			
Requirement implementation	Static	45	22.5
Code execution	Dynamic	40	20.0
Data authenticity	Dynamic	15	7.5
Code quality $\lambda_c = 35\%$			
Readability	Static	15	7.5
Robustness	Static	20	10.0
Time/space complexity	Static	10	5.0
Repair cost	Static	10	5.0
Architecture and technical choice	Static	15	7.5
User experience $\lambda_c = 15\%$			
Interface and interaction	Dynamic	15	7.5
Information presentation	Dynamic	10	5.0
Feature richness	Static	5	2.5

The relative priorities in Table 8 are normalized separately within each group. Requirement implementation receives $0.50 \times 45/100 = 0.225$ and code execution receives $0.50 \times 40/100 = 0.20$, preserving the expert priorities discussed in RQ2.1. These final weights govern the reassessed model Total in Table 1, connecting expert judgments of user needs to the reported aggregate scores. Appendix A.3.2 specifies their application to each artifact and the subsequent model-level average.

Evidence requirements determine how each criterion is assessed. Seven dimensions can be judged through source inspection. Code execution, data authenticity, interface and interaction, and information presentation additionally require runtime or rendered evidence. Relevant observations may inform more than one dimension, with dimension-specific overview instructions in Table 9.

Derive agent scoring points from the human rubric. We first iterate the human score anchors until annotators achieve high agreement, then fix the rubric dimensions, weights, and human reference judgments. The judge agent initially receives this human rubric directly, but its scores do not align sufficiently with the human scores. We therefore iteratively revise the agent's scoring points over multiple rounds, evaluating each revision against the same human reference while keeping the dimensions and weights unchanged. The final agent anchors result from this calibration process.

Table 9: Overview instructions for the eleven rubric dimensions. The complete human and agent score anchors are available in the anonymous repository.

Dimension	Overview instruction
Requirement implementation	Map every core requirement to its implementation location or identify its absence.
Code execution	Use actual startup, interaction, service-response, and output evidence.
Data authenticity	Trace material outputs to their input, parser, computation, or service, and compare with observed results where available.
Readability	Cite naming, control flow, decomposition, and specific code passages from the perspective of a developer taking over the artifact.
Robustness	Inspect error branches, input validation, resource lifetimes, and recovery behavior on core paths. Identify what happens after an exception is caught, including silent suppression or useful handling.
Time/space complexity	Inspect the main algorithm and high-frequency paths, explaining how time or memory grows with the workload.
Repair cost	First identify concrete remaining defects, then assess the affected code, dependencies, and scope of repair.
Architecture and technical choice	Describe the actual modules, dependencies, data flow, and technology choices.
Interface and interaction	Inspect actual rendered screenshots together with interaction evidence. Examine layout, entry points, action feedback, and completion of core user operations.
Information presentation	Inspect rendered information hierarchy, grouping, text and chart legibility, and adaptation to the display.
Feature richness	Separate explicitly requested capabilities from useful additions, such as filtering, search, statistics, or import and export. List each addition and locate its implementation, distinguishing working behavior from decorative controls and placeholders.

Table 9 summarizes the overview instructions; the complete human and agent score anchors are provided in the anonymous repository. Both retain the same three-level quality scale. The additional detail in the agent anchors reflects clarifications introduced to address observed disagreements, including what evidence to inspect, how to distinguish severity levels, and where dimension boundaries lie. Thus, the agent scoring points are calibrated instructions for applying the established human standard.

Human annotators can use professional experience and task context to interpret broad anchors, including whether a defect is consequential or a design fits the project. The agent needs more detailed guidance on these judgments, with explicit evidence requirements, severity thresholds, and boundary cases. These instructions help it apply the same quality standard without relying on superficial cues or counting one defect across multiple dimensions. To assess the calibrated rubric, we compare the two human annotators, the agent, and GT on the same artifacts; Table 13 in Appendix A.4.4 presents these agreement comparisons.

Apply evidence requirements and dimension boundaries consistently. Each grade must be supported by source, runtime, or visual evidence under the dimension-specific instructions in Table 9. The judge agent records missing evidence, distinguishes evaluation failures from artifact defects, and respects dimension boundaries to avoid penalizing the same defect repeatedly.

First stabilize human judgments. All annotators hold a bachelor's degree in a computer-related discipline and have at least two years of software development experience. The first two human annotation rounds are carried out independently by Human A and Human B on the same calibration artifacts. Each annotator applies the human rubric and records a score and supporting rationale for every artifact–dimension pair. We compare their scores and rationales to identify ambiguities in the anchors, revise the human rubric, and repeat independent assessment until high agreement is reached.

Establish GT through a third human annotation round. With the human rubric fixed, a third annotator reviews the first two annotators' scores and scoring rationales alongside the artifact itself.

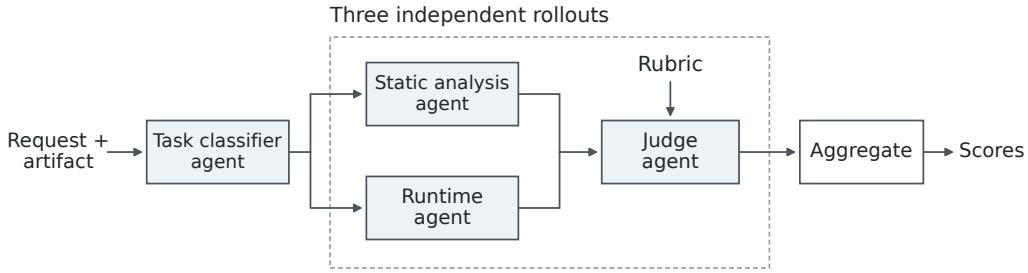


Figure 10: Artifact scoring workflow. A task classifier agent selects the evaluation setting. In each of three independent rollouts, static analysis and runtime agents collect evidence, and a judge agent assigns rubric-based grades. These grades are aggregated into artifact scores and measures of agreement across rollouts.

This review integrates the earlier assessments with direct inspection to assign the final reference score $g_{i,d}$ for artifact i on dimension d. These third-round human scores constitute GT. The original independent scores and rationales remain available as Human A and Human B, supporting both direct inter-annotator comparisons and comparisons with the final human reference.

Then refine agent scoring points against the human reference. Starting from the human rubric, we repeatedly compare the judge agent's grades and rationales with GT and refine its evidence requirements, severity thresholds, and dimension boundaries. The dimensions, weights, and human reference remain fixed throughout; only the agent's scoring points change. Table 9 summarizes the resulting inspection instructions. Appendix A.3.2 defines the agreement measures, while Table 13 and Figure 11 report the human–human, human–agent, and GT-referenced comparisons.

A.3.2 Evidence Collection and Score Aggregation

The scoring workflow applies the calibrated agent criteria described in Appendix A.3.1 to a candidate artifact. Figure 10 separates evidence collection, rubric-based judgment, and numerical aggregation. Static analysis and runtime agents collect evidence, which the judge agent uses to assign dimension-level grades under the fixed rubric. GLM-5.3-Flash serves as the judge agent for the model-panel artifacts. Deterministic aggregation converts repeated grades into an artifact score and diagnostics of agreement.

Collect evidence for the rubric dimensions. A task classifier agent uses the request and artifact structure to select the task-specific rubric and execution setting. A browser, application, service, script, or desktop workflow supplies observations appropriate to the task. A static analysis agent identifies implemented functions, missing components, and code relevant to the quality criteria. A runtime agent exercises the artifact and records outputs, failures, interaction states, and screenshots. It can configure the execution environment but leaves the candidate's source unchanged.

Source evidence supports the seven static dimensions in Table 8; execution and rendered evidence additionally support code execution, data authenticity, interface and interaction, and information presentation. The two evidence sources are interpreted together. For example, finding an event handler establishes that an interaction is implemented, while exercising the control establishes whether it works for the user. The judge agent also distinguishes an observed artifact defect from a failure to obtain the required evidence, following the calibrated score anchors described in Appendix A.3.1.

Repeat observation and judgment independently. Let i index an evaluated artifact and d a rubric dimension. For each artifact, we use $K = 3$ independent scoring rollouts, indexed by $k \in \{1, \ldots, K\}$. Each rollout collects its own source and runtime evidence without access to the observations or grades of the other rollouts. Thus, repetition captures variation in both the evidence found and its interpretation.

Following §3.3, $e_{i,d}^{(k)}$ denotes the evidence relevant to dimension d in rollout k, and R_d is its fixed agent-facing specification, including the score anchors and evidence requirements. The judge agent J assigns

$$\hat{h}_{i,d}^{(k)} = J\big(R_d, e_{i,d}^{(k)}\big) \in \{0, 1, 2\}, \tag{10}$$

where a larger grade indicates stronger delivery. Each grade is accompanied by a rationale connecting specific observations to the applicable anchor. R_d is held constant across scoring rollouts; the rubric refinement described in Appendix A.3.1 occurs during calibration, not while judging a benchmark artifact.

Separate the mean score, comparison label, and self-consistency. Let $\mathbb{K}_{i,d} \subseteq \{1, \ldots, K\}$ contain the indices of rollouts that return a valid grade for artifact i on dimension d, and let $K_{i,d} = |\mathbb{K}_{i,d}|$ be their number. When $K_{i,d} > 0$, aggregation gives

$$\mu_{i,d} = \frac{1}{K_{i,d}} \sum_{k \in \mathbb{K}_{i,d}} \hat{h}_{i,d}^{(k)}, \tag{11}$$

$$a_{i,d} = \mathrm{round}(\mu_{i,d}), \tag{12}$$

$$\mathrm{SC}_{i,d} = \frac{1}{K_{i,d}} \max_{v \in \{0,1,2\}} \sum_{k \in \mathbb{K}_{i,d}} \mathbf{1}\{\hat{h}_{i,d}^{(k)} = v\}. \tag{13}$$

Here $\mu_{i,d} \in [0, 2]$ is the mean used to compute artifact quality, $a_{i,d} \in \{0, 1, 2\}$ is the rounded label used for comparison with human judgments, and $\mathrm{SC}_{i,d}$ is the fraction of available rollouts assigning the most frequent grade. The indicator $\mathbf{1}\{\cdot\}$ equals one when its condition holds and zero otherwise. Rounding uses the nearest integer, with ties to even.

For example, grades $(2, 2, 1)$ give a mean of $5/3$, a comparison label of 2, and self-consistency of $2/3$. The mean preserves variation between grades, whereas self-consistency describes their reproducibility. A consistently repeated grade can still disagree with humans, so self-consistency does not replace human-reference calibration. If no rollout supplies a valid grade for a dimension, that dimension remains unscored.

Aggregate dimensions into artifact and model scores. Model comparison uses the final expert-informed weights w_d in Table 8, as applied in the reassessment underlying the reported Total scores. Let $\mathbb{D}_i$ be the nonempty set of applicable dimensions with valid mean scores for artifact i. Its normalized score is

$$s_i = \frac{\sum_{d \in \mathbb{D}_i} w_d \, \mu_{i,d}/2}{\sum_{d \in \mathbb{D}_i} w_d} \in [0, 1]. \tag{14}$$

Dividing by two places each dimension on the unit interval; the denominator renormalizes the final weights over the applicable, scored dimensions while preserving their relative priorities. When all eleven dimensions apply and are scored, the denominator is one and accuracy, code quality, and user experience retain their group weights of 50%, 35%, and 15%. Group summaries use the same weighted calculation restricted to the corresponding dimensions. Reported artifact and group scores multiply the normalized values by 100.

For model m, let $\mathbb{A}_m$ contain the indices of its scored artifacts, with one artifact per evaluated task, and let $n_m = |\mathbb{A}_m|$. For $n_m > 0$, its mean normalized score and reported score are

$$\bar{s}_m = \frac{1}{n_m} \sum_{i \in \mathbb{A}_m} s_i, \qquad \mathrm{Score}(m) = 100\bar{s}_m. \tag{15}$$

Each scored task therefore contributes equally to the model average. Appendix A.4.6 describes how model-panel results are consolidated across compute nodes, and Table 14 reports dimension-level scores under this aggregation rule. Appendix A.4.6 compares quality with execution cost on common task sets. Revision comparisons match tasks and starting artifacts across rounds as described in Appendix A.5.

Compare the automated label with the human reference. The reference $g_{i,d}$ is the third-round human annotation defined in Appendix A.3.1. For a dimension d, let $\mathbb{P}_d$ contain the indices of

artifacts with both an automated comparison label $a_{i,d}$ and a human reference $g_{i,d}$, and let $N_d = |\mathbb{P}_d|$. For $N_d > 0$, exact agreement is

$$\text{Exact}_d = \frac{1}{N_d} \sum_{i \in \mathbb{P}_d} \mathbf{1}\{a_{i,d} = g_{i,d}\}. \tag{16}$$

Within-one agreement replaces the equality condition with $|a_{i,d} - g_{i,d}| \leq 1$. Signed bias is the mean of $a_{i,d} - g_{i,d}$ over the same paired artifacts, with a positive value indicating more generous automated grades. Usability agreement compares $\mathbf{1}\{a_{i,d} > 0\}$ with $\mathbf{1}\{g_{i,d} > 0\}$ to measure agreement on the distinction between inadequate and at least partially usable delivery.

Cohen's $\kappa_d = (p_{\text{o},d} - p_{\text{e},d})/(1 - p_{\text{e},d})$ additionally adjusts for chance agreement. Here $p_{\text{o},d} = \text{Exact}_d$, and $p_{\text{e},d}$ is the sum, over score categories, of the product of the agent and human-reference category proportions on $\mathbb{P}_d$. It is undefined when $p_{\text{e},d} = 1$. This adjustment helps distinguish agreement produced by matching individual judgments from agreement driven by assigning one common grade to most artifacts.

Human–GT comparisons apply the same metrics separately to each annotator's original grades, then average the two annotator-level results. Agent–GT uses $a_{i,d}$ against the same reference. These comparisons implement the human-alignment objective in Appendix A.3.1, while self-consistency remains a diagnostic of repeated automated scoring. Weighted agreement summaries use the same final expert-informed weights as artifact scoring (Appendix A.3.1), applying them to dimension-level agreement statistics rather than artifact grades. Table 13 reports the resulting chance-corrected comparisons.

A.4 Supplementary Validation and Agent Results

This section supplements the main experiments with component-level validation and extended analyses. Appendix A.2 defines how recovery and trajectory-equivalence checks operate. Here we explain how the recovered files and environments are assessed, which cases enter each measurement, and what the observed results establish. The workspace recovery study checks recovered inputs directly, while the environment studies examine deployment feasibility, replay outcomes, and the contribution of repair. Subsequent analyses assess judge agent agreement, explain how recovered tasks support candidate-agent execution and artifact scoring, and describe model-result aggregation.

A.4.1 Workspace recovery calibration

Identify cases that need file recovery. We use a calibration set of 100 cases to assess recovery quality. Human annotators first label whether each task requires pre-existing local files, and the agent independently makes the same decision from the request and trace. Their decisions agree on 96 of the 100 cases; 57 cases require file recovery.

Reconstruct the files and assess their quality. The agent reconstructs the workspaces for these 57 cases, after which human annotators assess whether the required inputs and observed contents are restored faithfully. They check completeness, unsupported file generation, and the inclusion of files created during the original solution. Human reviewers accept 82.6% of the assessable reconstructions as successful. Table 10 reports the assessed case counts, acceptance rates, and file-quality findings; Table 11 summarizes workspace and environment recovery results.

A.4.2 Environment acceptance and repair studies

Check feasibility, deployment, and replay outcomes. We select a calibration set of 143 cases from the 14,603 coding traces to assess environment recovery, using the final replay score to evaluate the recovered workspace and environment jointly. After checking platform and hardware feasibility, we build and launch each supported environment with $\widehat{W}_0$ and replay the recorded tool calls. This yields 132 scored reconstructions, with ten cases excluded for unsupported platform or hardware requirements and one assessment incomplete.

Here, trajectory equivalence is quantified by the replay score in Eq. 3: a score of 1 over the complete tool-call sequence indicates that the recovered workspace and environment reproduce the original

Table 10: Human assessment of workspace recovery. Agreement on recovery need compares agent and human decisions; reconstruction success measures acceptance of the recovered files. Each rate uses the cases assessed for that criterion, with a Wilson 95% confidence interval (CI).

Assessment criterion	Cases	Rate (%)	95% CI
Agreement on recovery need	96/100	96.0	[90.2, 98.4]
Reconstruction success	38/46	82.6	[69.3, 90.9]
Success excluding binary limitations	38/43	88.4	[75.5, 94.9]
Unsupported file generation	0/41	0.0	[0.0, 8.6]
Inclusion of solution-created files	1/41	2.4	[0.4, 12.6]

Table 11: Workspace and environment recovery results. Workspace quality is assessed by human review; environment quality is assessed through the replay score, which measures agreement with the original tool-call results. Denominators indicate the cases assessed for each metric.

Layer	Metric	Denominator	Result
Workspace	Need-recovery agreement	100	96/100 (96.0%)
	Reconstruction success	46	38/46 (82.6%)
	Success excluding binary limitations	43	38/43 (88.4%)
	Unsupported file generation	41	0/41 (0.0%)
Environment	Successfully reconstructed and scored	143	132/143 (92.3%)
	Mean replay score	132	0.9175
	Replay score ≥ 0.8	132	119/132 (90.2%)

recorded results. Let $\mathbb{V}$ index the 132 scored recovery cases and $v_j \in [0, 1]$ the final replay score for case $j \in \mathbb{V}$. We summarize both the average score and the fraction of cases reaching 0.8,

$$\bar{v} = \frac{1}{|\mathbb{V}|} \sum_{j \in \mathbb{V}} v_j = 0.9175, \qquad \frac{|\{j \in \mathbb{V} \mid v_j \geq 0.8\}|}{|\mathbb{V}|} = \frac{119}{132} = 90.2\%. \tag{17}$$

These results show that most recovered environments reproduce the recorded tool behavior closely.

Compare construction, launch repair, and full recovery. To assess the contributions of cold-start repair and replay-guided refinement, we compare the three conditions in Table 12. The launcher agent introduced in Appendix A.2.2 handles build and startup failures; the full workflow also uses temporal dependency inference and replay evaluation to guide environment tuning.

For the 29 cases evaluated under +Launcher, we define difficulty using the replay score s: Easy ($s \geq 0.95$), Medium ($0.70 \leq s < 0.95$), and Hard ($s < 0.70$). Full recovery contributes its 26 available scored cases; differences compare condition-level means within these groups.

Full recovery raises the reported mean from 0.5303 to 0.8209 in the hard stratum and from 0.8461 to 0.9165 in the medium stratum, while the easy-stratum mean falls from 0.9874 to 0.9320. This pattern locates the main benefit in cases where launch repair still leaves behavioral discrepancies, while the decline in easy cases shows that further repair can reduce replay agreement.

Assess replay-guided refinement. Within the environment-recovery calibration set, cases with both pre-refinement and final scores improve from a mean replay score of 0.8192 to 0.9175. This result supports replay-guided refinement as a means of improving recovery quality. Together, these studies assess deployment feasibility, replay fidelity, and the gains from refinement.

A.4.3 Human and judge agent calibration

The calibration study uses the same 56 generation artifacts and adjudicated GT to compare automated grades with two independent human annotators. Figure 11 reports dimension-level exact agreement and highlights differences between the judge agent and the human average. This comparison identifies strengths and weaknesses across rubric dimensions.

Table 12: Replay scores under three environment-recovery conditions. Single-shot makes one construction attempt, counting build failures as zero. +Launcher adds an agent that builds, cold-starts, and repairs the environment. Full also includes dependency inference and replay-guided refinement. Difference reports Full minus +Launcher.

Stratum	Single-shot	+Launcher	Full	Difference
Hard	.0750	.5303	.8209	+.2906
Medium	.0000	.8461	.9165	+.0704
Easy	.0000	.9874	.9320	-.0554

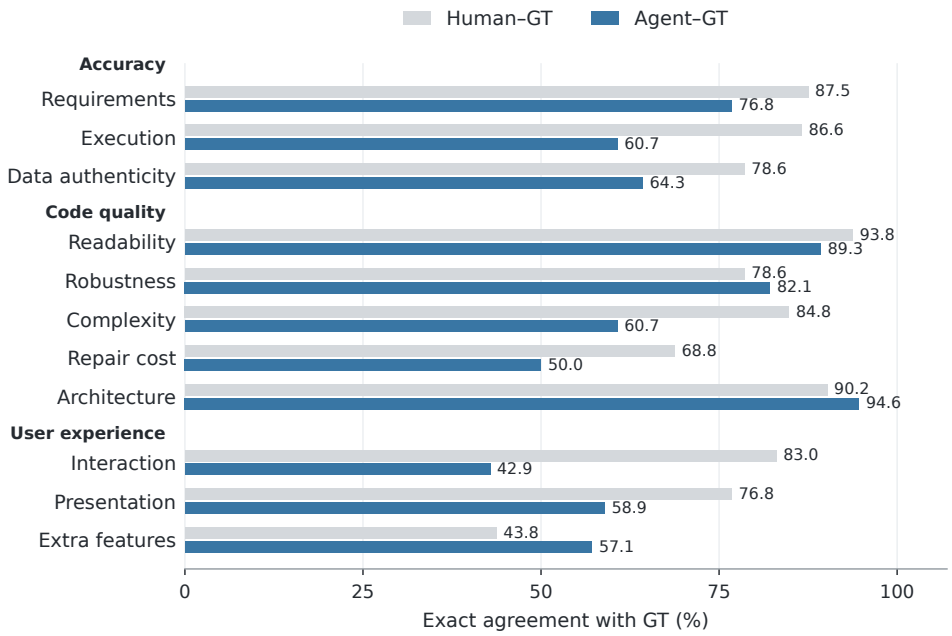


Figure 11: Exact agreement with the adjudicated human reference (GT) across rubric dimensions. Human–GT averages the two annotators' agreement rates. Agent–GT compares the judge agent's grade, obtained by rounding its mean over three scoring rounds, with the same reference.

Agreement varies substantially across dimensions, with lower judge agent accuracy on several criteria requiring runtime or visual interpretation as well as some source-based criteria. These differences motivate complementing exact agreement with chance-corrected measures.

A.4.4 Human–human, agent–human, and GT-referenced agreement

GT comes from a third human annotation round: an annotator reviews Human A's and Human B's scores and rationales together with the artifact itself to assign the final reference grades (Appendix A.3.1).

Table 13 distinguishes direct inter-rater agreement from agreement with adjudicated GT. Each automated grade is the rounded mean of three scoring rounds, yielding 616 artifact–dimension pairs. For each dimension, human–human κ compares Human A with Human B; agent–human κ averages the separate Agent–Human A and Agent–Human B values. Human–GT likewise averages the two annotators' separate comparisons with GT, while Agent–GT compares the agent directly with GT. Cohen's $\kappa = (p_o - p_e)/(1 - p_e)$ adjusts observed agreement p_o for the agreement p_e expected from the marginal grade distributions. Within-one agreement and signed bias provide complementary GT-referenced diagnostics.

The final weights follow expert priorities, are confirmed through reassessment, and are listed in Table 8 (Appendix A.3.1). For a dimension-level agreement statistic M_d, the overall summary

Table 13: Agreement between human annotators (H–H), between the judge agent and human annotators (A–H), and with the adjudicated human reference (GT). Cohen's κ measures agreement beyond chance. W-1 is the proportion of grades within one point of GT; bias is the mean signed difference from GT, with positive values indicating higher grades.

	Direct comparison		Human–GT			Agent–GT		
Dimension	H–H κ	A–H κ	κ	W-1	Bias	κ	W-1	Bias
Accuracy								
Requirement implementation	0.303	0.075	0.532	0.946	−0.11	0.188	1.000	−0.09
Code execution	0.622	0.283	0.762	0.938	−0.04	0.287	0.946	+0.16
Data authenticity	0.312	0.265	0.589	0.955	−0.06	0.248	0.911	+0.16
Weighted group mean	0.432	0.187	0.632	0.944	−0.07	0.236	0.965	+0.05
Code quality								
Readability	0.000	−0.028	0.148	1.000	+0.03	−0.057	1.000	+0.00
Robustness	0.074	0.000	0.452	0.991	+0.01	0.000	1.000	−0.18
Complexity	0.407	−0.008	0.631	1.000	+0.06	−0.020	1.000	+0.14
Repair cost	0.160	0.064	0.425	0.973	−0.02	0.013	0.982	+0.12
Architecture	0.317	0.306	0.397	0.982	−0.07	0.387	1.000	+0.03
Weighted group mean	0.170	0.068	0.397	0.990	−0.00	0.070	0.997	−0.01
User experience								
Interface and interaction	0.527	0.130	0.732	0.938	−0.05	0.138	0.946	+0.12
Information presentation	0.366	0.288	0.578	0.920	−0.12	0.324	1.000	−0.02
Feature richness	0.289	0.113	0.123	0.705	−0.70	0.135	0.911	−0.23
Weighted group mean	0.434	0.180	0.579	0.893	−0.18	0.200	0.958	+0.02
Weighted macro mean	0.341	0.144	0.542	0.952	−0.06	0.173	0.975	+0.02

is $\sum_d w_d M_d$; a group summary normalizes the weights over that group's dimensions. Thus, the reported weighted κ averages dimension-level κ values, emphasizing agreement on the aspects of quality that experts prioritize for users.

Overall agreement. On individual criteria, the judge agent can exceed the human average in both exact agreement and κ, as observed for feature richness (Figure 11 and Table 13). These calibration results support partial replacement of manual annotation with calibrated agent scoring. Because our evaluation centers on user experience, human review remains important for nuanced, context-dependent quality judgments. Appendix A.4.6 specifies the judge agent used for the model panel.

A.4.5 From recovered tasks to model rollout and artifact scoring

This section details the execution and scoring protocol for the R1/R2 comparison in §4.4, reported in Table 1 and Figure 4.

Execute the request to produce the R1 artifact. Each task supplies the constructed user request q, recovered workspace $\widehat{W}_0$, and recovered initial environment $\widehat{E}_0$ defined in Appendix A.2. We package these inputs for the Terminal-Bench 2.0 execution infrastructure (Merrill et al., 2026), using our reconstructed tasks. Claude Code and OpenHands serve as two agent harnesses through which candidate models read files, edit code, and execute commands under the same evaluation protocol. Each initial attempt starts from a fresh copy of the task context, allowing the candidate to inspect the available inputs and develop its own solution. The execution infrastructure retains the resulting files and action record for evaluation. A small number of runs affected by API rate limits or environment failures are retried before their artifacts enter the same scoring and pairing procedure used for both harnesses.

Score the R1 artifact. Let y denote the artifact produced by this execution, comprising the files created or modified to address q. For a software-development request, successful delivery means that the project can run and provide the requested behavior in the task environment. For a request targeting scripts, data, or other files, the delivered directory should contain the requested outputs and

Table 14: Dimension-level breakdown of the Table 1 results under Claude Code.

Model	Round	Accuracy			Code quality					User experience		
		Req.	Exec.	Data	Read.	Rob.	Complex.	Repair	Arch.	Interact.	Present.	Richness
DeepSeek	R1	76.2	74.2	74.9	83.3	81.1	82.9	81.5	83.1	64.8	66.4	67.6
	R2	98.3	96.3	97.0	94.1	91.9	93.7	92.3	93.8	88.0	89.6	90.8
GLM-5.2	R1	80.9	78.9	79.6	87.8	85.6	87.4	86.0	87.6	66.9	68.5	69.7
	R2	95.9	93.9	94.6	97.4	95.2	97.0	95.6	97.1	86.5	88.1	89.3
GLM-5.3	R1	84.6	82.6	83.3	91.6	89.4	91.2	89.8	91.4	68.6	70.2	71.4
	R2	98.2	96.2	96.9	100.0	98.1	99.6	98.4	99.8	92.0	93.6	94.8
GPT-5.5	R1	77.9	75.9	76.6	87.6	85.4	87.2	85.8	87.4	57.9	59.5	60.7
	R2	90.2	88.2	88.8	96.8	94.6	96.4	95.0	96.5	76.8	78.4	79.6
GPT-5.6 sol	R1	87.0	85.0	85.7	93.4	91.2	93.0	91.6	93.1	73.4	75.0	76.2
	R2	95.5	93.5	94.1	98.3	96.1	97.9	96.5	98.0	87.0	88.6	89.8
Kimi-K3	R1	79.5	77.5	78.2	88.9	86.7	88.5	87.1	88.6	58.8	60.4	61.6
	R2	89.4	87.4	88.0	94.8	92.6	94.4	93.0	94.5	75.0	76.6	77.8
MiniMax-M3	R1	76.9	74.9	75.6	86.3	84.1	85.9	84.5	86.1	57.4	59.0	60.2
	R2	92.0	90.0	90.6	97.2	95.0	96.8	95.4	96.9	80.7	82.3	83.5
LongCat	R1	83.3	81.3	82.0	91.8	89.6	91.4	90.0	91.5	62.9	64.5	65.7
	R2	92.1	90.1	90.7	97.1	94.9	96.7	95.3	96.8	78.3	79.9	81.1
Fable 5	R1	93.4	91.4	92.1	97.0	94.8	96.6	95.2	96.7	80.3	81.9	83.1
	R2	98.4	96.4	97.0	100.0	98.3	99.7	98.6	99.8	89.5	91.1	92.3

support the operations needed to verify them. The scoring workflow in Appendix A.3.2 combines source inspection with execution or rendered evidence and applies the criteria in Appendix A.3.1 to produce dimension scores, supporting rationales, and an aggregate artifact score.

Revise and rescore the artifact in R2. The R1 artifact and its scoring evidence feed the revision workflow in Appendix A.5, which turns identified weaknesses into an instruction. In R2, the same candidate model follows this instruction to revise its preserved R1 artifact. We score the revised artifact under the same rubric and aggregation rule and compare both rounds on matched tasks. This produces the paired scores in Table 1 and the improvement, tie, and regression outcomes in Table 2, testing whether rubric-guided feedback improves the delivered artifact.

A.4.6 Model-panel configuration and repeated sampling

The model comparison reports reassessed Total scores under the final expert-informed weighting scheme in Table 8. Each artifact's score is a weighted mean over applicable dimensions, and the model Total averages these scores equally across tasks (Appendix A.3.2). Task identities are matched to the canonical benchmark set $\mathbb{B}$, and GLM-5.3-Flash supplies the judge agent for the model panel. Table 14 expands the generation dimensions underlying Table 1. Paired revision columns use the matched subsets defined in Appendix A.5.

Model references. Official model cards and release documentation describe DeepSeek-V4-Pro (DeepSeek, 2026), GLM-5.2 and GLM-5.3 (Z.ai, 2026a;b), GPT-5.5 and GPT-5.6 sol (OpenAI, 2026b;c), Kimi-K3 (Moonshot AI, 2026), MiniMax-M3 (MiniMax, 2026), LongCat-2.0 (Meituan, 2026), Fable 5 (Anthropic, 2026), and Gemini 3.6 Flash (Google, 2026a). Table 1 links each model name directly to its documentation.

Gemini compatibility with the Claude Code harness. In our Claude Code deployment, the Anthropic-compatible gateway does not support Gemini's `thought_signature` metadata, a limitation identified during Gemini 3.6 Flash integration for multimodal evaluation. Gemini uses these signatures to preserve reasoning context across tool calls and requires them to be returned in subsequent requests (Google, 2026b). We therefore exclude Gemini–Claude Code results from the reported comparisons to avoid confounding artifact quality with this protocol incompatibility. Gemini–OpenHands results remain included.

Table 15: Dimension-level breakdown of the Table 1 results under OpenHands.

Model	Round	Accuracy			Code quality					User experience		
		Req.	Exec.	Data	Read.	Rob.	Complex.	Repair	Arch.	Interact.	Present.	Richness
DeepSeek	R1	89.0	87.0	87.6	92.9	90.7	92.5	91.1	92.7	73.5	75.1	76.3
	R2	93.2	91.2	91.9	98.5	96.3	98.1	96.7	98.3	88.4	90.0	91.2
Gemini 3.6	R1	83.7	81.7	82.3	91.1	88.9	90.7	89.3	90.8	69.2	70.8	72.0
	R2	92.1	90.1	90.8	97.0	94.8	96.6	95.2	96.8	84.8	86.4	87.6
GLM-5.2	R1	88.0	86.0	86.7	94.0	91.8	93.6	92.2	93.7	66.1	67.7	68.9
	R2	95.3	93.3	94.0	99.6	97.4	99.2	97.8	99.3	87.1	88.7	89.9
GLM-5.3	R1	83.2	81.2	81.9	91.1	88.9	90.7	89.3	90.9	70.6	72.2	73.4
	R2	91.5	89.5	90.2	96.0	93.8	95.6	94.2	95.8	77.1	78.7	79.9
GPT-5.5	R1	85.2	83.2	83.9	91.1	88.9	90.7	89.3	90.8	64.0	65.6	66.8
	R2	94.4	92.4	93.1	97.0	94.8	96.6	95.2	96.7	86.9	88.5	89.7
GPT-5.6 sol	R1	89.3	87.3	88.0	94.5	92.3	94.1	92.7	94.2	76.7	78.3	79.5
	R2	96.5	94.5	95.1	99.3	97.1	98.9	97.5	99.0	91.9	93.5	94.7
Kimi-K3	R1	78.2	76.2	76.8	87.8	85.6	87.4	86.0	87.5	58.0	59.6	60.8
	R2	87.8	85.8	86.4	93.7	91.5	93.3	91.9	93.4	74.8	76.4	77.6
MiniMax-M3	R1	85.4	83.4	84.0	91.6	89.4	91.2	89.8	91.3	64.4	66.0	67.2
	R2	91.2	89.2	89.9	99.0	96.8	98.6	97.2	98.7	84.6	86.2	87.4
LongCat	R1	79.8	77.8	78.5	89.1	86.9	88.7	87.3	88.9	60.0	61.6	62.8
	R2	89.3	87.3	88.0	94.9	92.7	94.5	93.1	94.6	76.9	78.5	79.7
Fable 5	R1	90.0	88.0	88.6	94.0	91.8	93.6	92.2	93.7	75.1	76.7	77.9
	R2	96.2	94.2	94.9	98.5	96.3	98.1	96.7	98.2	91.6	93.2	94.4

Aggregate repeated samples for cost and quality comparison. Experiments are distributed across multiple groups of compute nodes, and their outputs are consolidated by benchmark task identity to complete coverage of the original 129-task Trace2Eval benchmark. We collect three samples for each configuration and assess variability across these samples, applying the same task-specific scoring and aggregation rules throughout. Matching records by task identity provides a common basis for comparing model configurations across distributed runs. Figure 3 presents the first-round comparison under Claude Code, relating artifact quality to candidate-model execution cost. This joint view complements the capability-level scores by showing how the quality of delivered artifacts relates to the resources used to produce them.

A.4.7 Feedback ablation: an unguided second development pass

Hold the starting artifact fixed. To distinguish the benefit of rubric-derived feedback from another development opportunity, we compare two second-round conditions from the same R1 artifact and recovered environment. The feedback condition follows Appendix A.5; the unguided condition receives the original user request verbatim, without weaknesses, scoring rationales, or additional improvement instructions. The candidate model, recorded sampling settings, and execution limits are matched between the two conditions. We compare the latest available attempt per task and retain the common set with verified agent participation and complete three-rollout scoring in all three conditions.

Compare Total scores on the same tasks. Table 16 reports the detailed values underlying Figure 5. Each row lists the R1 baseline, the unguided second pass (U2), the feedback-guided second pass (R2), and the corresponding gains over R1.

Feedback adds value beyond another development pass. An unguided second pass improves the mean in every reported configuration, confirming that an additional development opportunity already has value. Rubric-guided revision achieves a higher mean than the unguided pass across the detailed comparisons in Table 16.

These results complement the qualitative cases in Appendix A.5.2: the feedback identifies specific defects that a repeated original request leaves implicit.

Table 16: Detailed values for Figure 5. R1 denotes initial generation; R2 is revision with rubric feedback, and U2 is revision using the unchanged original request. Both revisions start from the same R1 artifact and environment. Scores are % of maximum; ΔR2 and ΔU2 measure gains over R1, and all differences are in pp.

Model	R1	R2	U2	ΔR2	ΔU2	R2−U2
Claude Code						
DeepSeek-V4-Pro	76.85	96.13	82.63	+19.28	+5.78	+13.50
GLM-5.2	80.96	94.13	86.39	+13.17	+5.43	+7.74
GLM-5.3	84.76	97.29	90.20	+12.53	+5.44	+7.09
GPT-5.5	78.10	89.88	82.85	+11.78	+4.75	+7.03
GPT-5.6 sol	86.23	94.87	89.71	+8.64	+3.48	+5.16
Kimi-K3	78.93	88.07	82.62	+9.14	+3.69	+5.45
MiniMax-M3	76.64	91.57	83.61	+14.93	+6.97	+7.96
LongCat-2.0	82.65	92.04	86.44	+9.39	+3.79	+5.60
Fable 5	92.42	97.37	94.42	+4.95	+2.00	+2.95
OpenHands						
DeepSeek-V4-Pro	85.72	93.75	90.79	+8.03	+5.07	+2.96
Gemini 3.6 Flash	82.89	91.76	85.27	+8.87	+2.38	+6.49
GLM-5.2	84.27	95.09	88.75	+10.82	+4.48	+6.34
GLM-5.3	84.90	90.41	86.79	+5.51	+1.89	+3.62
GPT-5.5	82.86	94.62	88.19	+11.76	+5.33	+6.43
GPT-5.6 sol	87.46	95.89	91.28	+8.43	+3.82	+4.61
Kimi-K3	77.54	86.16	81.44	+8.62	+3.90	+4.72
MiniMax-M3	82.40	91.97	88.21	+9.57	+5.81	+3.76
LongCat-2.0	79.52	88.18	83.44	+8.66	+3.92	+4.74
Fable 5	88.98	95.91	92.12	+6.93	+3.14	+3.79

A.5 Rubric-Guided Revision and Analysis

A.5.1 Revision Protocol

This workflow provides a practical test of the rubric by examining whether its diagnoses guide artifact improvement (§4.4). R1 is the candidate's initial artifact from the original request; R2 is its feedback-guided revision by the same model and harness in the preserved task environment. Both use the same rubric. A separate unguided revision starts from the same R1 artifact using only the original request, allowing the ablation to examine the contribution of targeted guidance beyond another development pass (Appendix A.4.7).

Turn observed weaknesses into actionable guidance. As shown in Figure 12, feedback preparation uses the original request, artifact description, dimension scores, and supporting rationales. A diagnosis agent identifies evidence-backed weaknesses and strengths. A design agent translates the weaknesses into concrete actions, such as repairing a failed operation or completing a missing workflow, while stating which working behavior should be preserved. A review agent checks that the instruction remains faithful to the request, covers the identified priorities, and asks for feasible changes. Instructions that fail this check are revised before delivery. GLM-5.2 performs these preparation roles, and the candidate model carries out the resulting instruction.

The instruction describes what to inspect, the intended improvement, and how to verify it. It directs the candidate toward observed defects while leaving implementation choices to that candidate. The revision starts from its first-round artifact in the preserved task environment; reference implementations and adjudicated human scores are not supplied. This preserves the connection between a diagnosed weakness and the artifact being improved.

Compare each revision with its actual starting point. For task i and model m, let $s_{i,m}^{(1)}$ and $s_{i,m}^{(2)}$ denote normalized R1 and R2 scores under the same rubric and aggregation rule (Appendix A.3.2). Under both Claude Code and OpenHands, each R2 artifact is paired with the R1 artifact used to

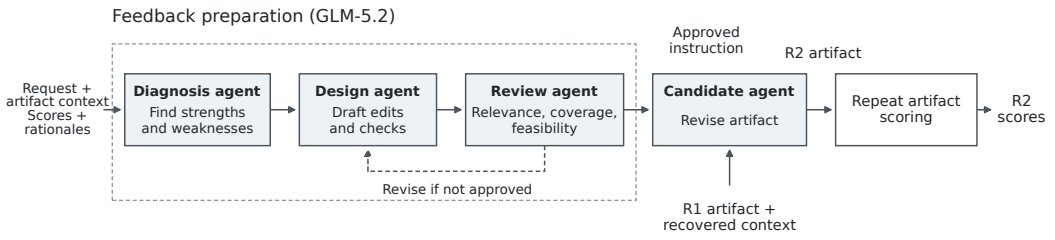


Figure 12: Rubric-guided feedback and artifact revision. Diagnosis and design agents identify weaknesses and propose edits; a review agent checks the instruction and returns it for revision when needed. The candidate agent applies the approved instruction to its R1 artifact. The revised artifact is scored using the same rubric and the workflow in Figure 10.

generate its feedback. Let $\mathbb{P}_m^{\text{rev}}$ contain the indices of matched tasks with preserved starting artifacts, completed feedback preparation, candidate participation, and complete paired scoring. For $|\mathbb{P}_m^{\text{rev}}| > 0$, we compute per-task differences and express the mean gain in percentage points (pp):

$$\Delta_{i,m} = s_{i,m}^{(2)} - s_{i,m}^{(1)}, \qquad G_m = \frac{100}{|\mathbb{P}_m^{\text{rev}}|} \sum_{i \in \mathbb{P}_m^{\text{rev}}} \Delta_{i,m}. \tag{18}$$

The sign of $\Delta_{i,m}$ identifies improvement, a tie, or regression. Means and outcome proportions use the same matched tasks. Under both the Claude Code and OpenHands harnesses, paired scoring requires scores for all applicable dimensions from three judge agent rollouts for each artifact in R1 and R2. We obtain 95% confidence intervals from the 2.5th and 97.5th percentiles of mean changes across 10,000 bootstrap resamples of matched task pairs.

Interpret the changes through the rubric. Table 1 and Figure 4 summarize paired score changes; Table 2 reports task outcomes. The unguided comparison in Appendix A.4.7 evaluates the contribution of this guidance beyond a second development pass.

A.5.2 Qualitative study of feedback-guided revision

We inspect matched R1/R2 artifacts together with their feedback instructions, source changes, and recorded execution evidence. The examples below illustrate how a diagnosed weakness becomes an edit and how fully that edit improves the delivered workflow.

Completing an interface makes existing functionality accessible. A DeepSeek-V4-Pro workflow application initially contains backend APIs and client-side calls, but its front end remains a placeholder without routed pages or usable controls. Feedback identifies this missing layer and asks the model to expose the existing functionality through forms, navigation, and data views. The revision adds a router and pages for authentication, workflow management, monitoring, and task history. Recorded browser interactions confirm page rendering, form entry, and navigation. Interaction and information-presentation scores both rise from 0 to 2 across the three scoring rounds. The improvement comes from connecting existing implementation work to a visible, operable interface.

Repairing a blocking error unlocks executable behavior. A GLM-5.2 service cannot compile because of truncated source and inconsistent interfaces. Feedback directs the model to repair these defects and verify the login endpoint; R2 compiles and returns the requested JSON response. Appendix A.5.3 presents the original request, feedback, source changes, and the corresponding artifact-score increase of 51.52 pp.

Connecting state, persistence, and controls improves data authenticity. A GLM-5.3 finance application already renders polished pages, but hard-coded summaries and an ineffective expense form prevent meaningful data updates. Feedback asks for a shared data layer and explicit connections between form controls, stored records, and derived totals. The revision introduces shared state with

persistent storage and binds expense submission to the data update function. Source inspection confirms these connections, and the execution evidence records working navigation and form interactions. Here, feedback targets the data flow beneath an existing interface rather than its appearance.

Cross-component consistency determines whether a repair is complete. A MiniMax-M3 commerce application receives feedback to implement missing backend endpoints and replace placeholder dependencies. Its revision adds product, group-purchase, cart, and order controllers, and endpoint probes return structured business data. However, frontend requests and routes remain inconsistent with the backend: the initial page loads, but navigation and data retrieval still fail. The code-execution score remains at 1 in all three R2 scoring rounds. Completing components individually therefore does not ensure a usable end-to-end workflow.

Together, these cases locate the clearest observed benefits in concrete, verifiable defects. They also explain why models can show different gains: an initially blocked or incomplete artifact offers substantial room for improvement, while realizing that gain requires the model to carry the repair through the relevant files, dependencies, and interfaces. The outcome depends on both the initial defect and the completeness of the implemented revision.

A.5.3 Case Study: From a Failed Build to a Working Login Service

This case follows GLM-5.2 under Claude Code through the feedback-guided revision protocol in §4.4. The task is to add a callable JSON login endpoint to an existing Scala/Pekko HTTP application while preserving its DTOs, actor routing, and Spring integration. It therefore requires both local controller code and integration with a working service.

To develop this login service, GLM-5.2 first executes the original request in the recovered workspace and environment, producing the initial artifact R1. The judge agent evaluates this delivery, and the pipeline uses its scores and rationales to formulate a revision instruction. GLM-5.2 then applies that instruction to the preserved R1 artifact in the same task environment, producing the revised artifact R2. Both artifacts address the same request and are assessed with the same rubric. In this case, R1 supplies much of the structure but cannot run; feedback-guided R2 raises the artifact's Total score from 46.97 to 98.49 (+51.52 pp).

The actual two-round task instructions. The complete R1 and R2 instructions are provided in the anonymous repository, with the Chinese portions translated into English. Filenames, line references, and task-directory paths are retained.

R1 implements the interface but stops short of execution. The candidate uses `Read` to inspect the User DTO and Maven configuration, `Edit` to add JSON-routing dependencies, and `Write` to create the authentication controller and HTTP bootstrap. Its subsequent `Bash` compilation attempt encounters missing Maven and Java executables. It searches for the tools but finishes without installing them or verifying a running endpoint. The later scoring run, with the build tools available, exposes truncated source and incompatible actor interfaces. Thus, R1's favorable readability and complexity grades coexist with an execution grade of zero: the controller is present, but the requested behavior remains unusable.

Feedback changes the candidate's work sequence. The R2 trajectory begins by reading the named files and using `TaskCreate` to turn the feedback into repair and verification tasks. The candidate again encounters the missing toolchain, but now installs OpenJDK and Maven through `Bash` before compiling. Feedback supplies a concrete completion condition, a service that builds and answers login requests, and the observed work proceeds toward that condition.

Source inspection also guides how the instruction is implemented. The candidate searches the room-factory call sites, aligns the factory signature with the actual caller, and replaces the incomplete HTTP helper with the JDK HTTP client. After removing the old HTTP dependency, compilation reports unresolved HTTP references in two dependent files; the candidate updates those callers and recompiles. Later stability edits introduce two further type errors, which it also fixes before continuing. The resulting sequence combines rubric-derived repair priorities with fresh compiler feedback, carrying changes through the affected components.

Selected tool interactions expose the improvement mechanism. The following sequence comes from the candidate's own tool calls and returned observations, rather than its final self-report. Calls are abridged to show the relevant arguments; file paths are expressed relative to the task workspace.

1. **Inspect and localize.** `Read(HttpService.scala)`, `Read(RoomFactory.scala)`, and `Read(OutboundActor.scala)` inspect the feedback's named defects. A `Bash` search for `RoomFactory.createRoom` finds the actual two-user call, allowing a concrete interface repair.
2. **Enable executable checks.** `Bash(apt-get install ... openjdk-21-jdk-headless maven)` supplies the missing toolchain. The candidate can now obtain compiler diagnostics and use them to direct subsequent edits.
3. **Close the repair loop.** `Write` and `Edit` update the HTTP helper, its callers, and actor handlers. Repeated `Bash(mvn compile)` calls expose remaining type errors; after the follow-up edits, the build succeeds. A final `Bash(mvn clean compile)` also returns `BUILD SUCCESS`.
4. **Verify the requested behavior.** After launching the application, `Bash(curl ... /auth/ping)` returns a healthy status and `Bash(curl -X POST ... /auth/login)` returns the user's fields and session information. A subsequent check parses the response and confirms that `accessToken` differs from `sessionId`, directly testing another feedback item.
5. **Make the result inspectable.** `Write` adds a Vue login form, followed by a successful `Bash(npx vite build)`. When the development server hits a file-watcher limit, the candidate switches to build preview and captures the rendered page. The later scoring runs confirm the backend response and assess the added interface.

Table 17: Artifact quality before and after feedback-guided revision in the GLM-5.2 login-service case under Claude Code. Dimension scores average three judge agent rounds on a 0–2 scale. For this illustrative case, Total is the unweighted mean of the eleven dimensions, rescaled to 0–100. Changes are R2 minus R1, in rubric points for dimensions and pp for Total.

Rubric dimension	R1	R2	Change
Requirement implementation	1.00	2.00	+1.00
Code execution	0.00	2.00	+2.00
Data authenticity	1.00	2.00	+1.00
Readability	2.00	2.00	0.00
Robustness	1.00	2.00	+1.00
Computational complexity	2.00	2.00	0.00
Repair cost	1.33	2.00	+0.67
Architecture	1.67	2.00	+0.33
Interface and interaction	0.00	2.00	+2.00
Information presentation	0.00	2.00	+2.00
Feature richness	0.33	1.67	+1.33
Total (%)	46.97	98.49	+51.52

From targeted feedback to measured gains. Table 17 shows the largest dimension-level gains in code execution, interface and interaction, and information presentation, each rising from 0 to 2, while readability and complexity retain their maximal R1 grades. The trajectory explains this pattern: explicit defect locations focus inspection, preservation instructions retain useful structure, and required execution checks expose incomplete repairs. Toolchain availability and cross-file dependencies influence progress, while compiler diagnostics and endpoint responses provide immediate feedback for the next action. Removing the build blocker makes existing functionality usable; the minimal interface then makes that functionality visible to the user. This case illustrates how the rubric can turn quality judgments into concrete repair and verification steps, complementing the aggregate feedback ablation in Appendix A.4.7.

A.6 PROFILE DISTRIBUTION SUMMARY

The distribution analysis in Appendix A.1 characterizes online requests by their intended activity, application setting, technical work, and user role. These four complementary aspects define the profile taxonomy used for task selection in §3.1. Task type and technical domain each have broad and fine-grained levels, giving six axes with 171 category entries in total.

Broad task type (`x1`). The profile begins with what the user asks the agent to do. Axis `x1` groups the overall activity into ten category entries, providing a first-level account of the mix of requested work in online traffic.

Fine-grained task type (`x2`). Within these broad activity families, `x2` specifies the requested operation through 37 finer entries. This second level distinguishes specific task demands while retaining their connection to the broader activities represented by `x1`.

Usage scenario (`y`). The requested activity also has a context of use, captured by the usage-scenario axis `y`. Its 46 entries describe the application or business setting in which the result will be used, connecting the requested work to its practical purpose.

Broad technical domain (`z1`). Alongside this application context, `z1` identifies the broad technical field involved in fulfilling the request. Its ten entries characterize the knowledge and practices needed to complete the work.

Fine-grained technical domain (`z2`). These broad domains are further differentiated by `z2`, whose 51 entries capture more specific technical specializations. This second level preserves differences in the technical demands of requests that share the same `z1` label.

User role (`w`). The user-role axis `w` completes the profile by describing who requests the work and the capacity in which the result is needed. Its 17 entries capture the user's professional context. Combined with the activity, scenario, and technical-domain labels, this information allows the distribution comparisons to account for both the requested work and its intended use.

Table 18 lists the complete category taxonomy. The anonymous repository provides the corresponding online-traffic and SWE-bench distribution tables. Appendix A.1.2 explains how the labels and reporting denominators are used in the distribution comparisons.

Table 18: User-profile taxonomy derived from online coding-agent traffic. The six profile axes contain 171 categories; #Cats gives the number of categories on each axis. Category names are translated from Chinese.

Axis	Category	#Cats
`x1` - Task type L1	Task Execution Programming & Technology Information Retrieval Data Processing Analysis & Reasoning Content Creation Others Practical Guidance N/A Multimedia Processing	10
`x2` - Task type L2	Automation Orchestration & Monitoring FI Feature Implementation: build a new capability from scratch CU Code Understanding: read, trace, or explain existing code General Knowledge Query BF Bug Fixing: diagnose and repair a defect Real-world Task Agent	

Axis	Category	#Cats
	Anomaly Diagnosis & Log Tracing	
	FE Feature Enhancement: extend or improve an existing feature	
	Platform & Business Consulting	
	Others	
	CD Configuration & Deployment: environment config, build, or deploy	
	Timely News Query	
	Functional Verification & Acceptance	
	Data Analysis & Insight	
	Runtime Env Config & Self-healing	
	Content Review & Evaluation	
	Data Extraction & Cleaning	
	Functional Writing	
	Information Comparison & Recommendation	
	RF Refactoring: restructure code without changing behavior	
	Logical Reasoning & Computation	
	Operational Advice / Guide	
	TG Test Case Generation: write or extend test cases	
	N/A	
	Text Translation	
	Conversational Text	
	Data Labeling & Classification	
	Text Editing & Optimization	
	Planning & Decision-making	
	PO Performance Optimization: improve speed, memory, or efficiency	
	Skill Acquisition & Enhancement	
	Creative Writing	
	Image Generation & Processing	
	Tutoring & Teaching	
	Audio / Video Processing	
	Cross-platform Auth & Identity Mgmt	
	Others	
y - Usage scenario	Programming	
	Computers & Electronics	
	Finance	
	Jobs & Education	
	Business & Industrial	
	Games	
	Health – Others	
	Health – Medical Consultation	
	Health – Medical / Rehab Services	
	Science	
	Home & Garden – Housekeeping & Repair	
	Home & Garden – Life Services	
	Home & Garden – Others	
	Arts & Ent. – Immersive & Board Games	
	Arts & Ent. – Wellness & Bath	
	Arts & Ent. – Others	
	Arts & Ent. – Indoor Shows & Ent.	
	Autos & Vehicles	
	Food & Drink – Others	
	Food & Drink – Meals & Specialties	
	Food & Drink – Takeout	
	Food & Drink – Fresh & Ingredients	
	Law & Government	46
	Real Estate	
	Sports	
	Shopping – Group Buying	
	Shopping – Flash / Instant Retail	

Axis	Category	#Cats
	Shopping – Others	
	Travel & Trans. – Tourism & Vacation	
	Travel & Trans. – Transit & Ride-sharing	
	Travel & Trans. – Lodging & Hotels	
	Travel & Trans. – Others	
	Online Communities	
	Books & Literature	
	Beauty & Fitness – Sports & Fitness	
	Beauty & Fitness – Medical Beauty	
	Beauty & Fitness – Salon & Hair	
	Beauty & Fitness – Others	
	Reference	
	News	
	Pets & Animals	
	Hobbies & Leisure	
	Internet & Telecom	
	People & Society	
	Others	
	N/A	
`z1` - Technical domain L1	Software Engineering & Development	10
	Data & AI	
	Others	
	Security	
	Systems & DevOps	
	Games & Entertainment	
	Engineering & Industry	
	Scientific Research	
	N/A	
	Others	
`z2` - Technical domain L2	A.1 Coding	51
	A.2 Build & Compilation	
	A.3 Version Control	
	A.4 Web Development	
	A.5 Testing	
	A.6 Debugging	
	A.7 API & Integration	
	A.8 Document Processing	
	B.1 System Configuration	
	B.2 Network Management	
	B.3 Container & Virtualization	
	B.4 Logging & Monitoring	
	B.5 Package Management	
	B.6 File Operations	
	B.7 Automation	
	C.1 Data Processing	
	C.2 Data Analysis	
	C.3 Visualization	
	C.4 ML Training	
	C.5 ML Inference	
	C.6 Computer Vision	
	C.7 NLP	
	C.8 Data Engineering	
	C.9 DL Frameworks	
	D.1 Numerical Computing	
	D.2 Statistical Analysis	
	D.3 Physics Simulation	
	D.4 Bioinformatics	
	D.5 Earth Science	
	D.6 Astronomy	
	D.7 Chemistry	
	D.8 Simulation & Modeling	
	E.1 Network Security	

Axis	Category	#Cats
	E.2 Application Security	
	E.3 Cryptography	
	E.4 Data Security	
	E.5 Vulnerability Fix	
	F.1 Manufacturing	
	F.2 Energy Systems	
	F.3 Control Systems	
	F.4 Optimization	
	F.5 Planning & Scheduling	
	F.6 Financial Modeling	
	G.1 Game AI	
	G.2 Game Development	
	G.4 Multimedia	
	H.1 Mathematics	
	H.2 Personal Assistant	
	H.3 General Tools	
	N/A	
	Others	
`w` - User role	Software Developer	
	ML / Algorithm Engineer	
	Operations Product / Content / User / Merchant	
	Data / Business Analyst & Strategy	
	Ops / SRE	
	Product Manager	
	N/A	
	QA / Test Engineer	
	Others	17
	Marketing / Growth	
	UX/UI Designer	
	HR Recruiting	
	Finance / Legal	
	Business / BD / Sales	
	Project Manager	
	Admin Assistant / Secretary	
	PR / Communications	