AGENTIC AUTO-RUBRICS: VERIFIABLE RUBRICS FOR JUDGING MULTI-TURN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Judging whether a coding agent fulfills a user's request is challenging in multi-turn sessions. In multi-turn sessions, user requirements constantly change and replace earlier requests, while user reactions can unintentionally reveal task outcome and bias evaluation. We introduce **Agentic Auto-Rubrics (AAR)**, a framework that compiles a multi-turn interaction into a verifiable specification of what the agent is responsible for. Given reaction-masked user turns, AAR constructs the active requirements and compiles them into fine-grained atomic rubrics, which specify explicit evidence types and codebase locations. To increase the judge's efficiency, AAR compresses similar atomic rubrics into a concise set of high-level parent rubrics. An agentic verifier then inspects the agent's workspace through read-only tools, evaluating these rubrics from concrete file-based evidence and references. Across multi- and single-turn coding benchmarks, AAR consistently outperforms direct scoring judges and existing rubric-based verifiers. While baseline verifiers degrade as sessions lengthen, AAR maintains stable accuracy across session lengths and five judge models, showing that compiling evolving interactions into verifiable specifications enables robust evaluation of coding agents.

1 INTRODUCTION

As coding agents currently handle interactive, multi-turn tasks, evaluation must shift beyond single-turn code generation. Evaluating such agents requires verifying whether the outcome fulfills the user's overall intent. To address this, outcome-based verifiers assess agent performance using trajectories alongside the final codebase. These verifiers serve three main purposes: assessing performance without execution tests, selecting rollouts at test time, and providing training rewards (Cobbe et al., 2021; Brown et al., 2024).

Evaluating multi-turn interactive sessions presents three unique challenges. First, **requirements evolve dynamically**. Users continuously add constraints, refine prior commands, or supersede earlier instructions. Thus, the agent should be evaluated against the requirements active at session end, rather than the initial prompt or the full conversation. (Laban et al., 2026). Second, **user reactions introduce shortcut biases**. Cues like praise or complaints often leak whether a task succeeded, leading evaluators to score the user's reaction rather than verifying the actual code. Third, **ground truth is inherently absent**. Real-world sessions rarely have test suites or reference solutions. They also lack alternative agent trajectories, making it hard to evaluate counterfactual actions.

Rubrics are widely used to mitigate the unreliability of direct LLM judgments (Gunjal et al., 2026; Viswanathan et al., 2025; Liu et al., 2026; Wang & Xiong, 2025). However, because existing rubric generation methods are primarily designed for single-turn, they exhibit limitations when applied to multi-turn agent sessions:

1. *Static rubrics* extracted solely from the initial user prompt perform competitively on short interactions but quickly degrade as user requirements evolve over longer sessions.
2. *Trajectory-conditioned rubrics* tend to overfit to that particular rollout and cannot fairly evaluate alternative rollouts.
3. *Trajectory-only evaluation* produces rationales and scores based on trajectory alone, yielding judgments that lack verifiability against the final codebase.

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.

To overcome these limitations, we propose Agentic Auto-Rubrics (AAR), illustrated in Figure 1. Rather than inspecting the whole trajectory and workspace, AAR first extracts user requirements, filters out conversational feedback, and continuously updates the active requirements by removing outdated ones. Each requirement becomes an atomic, verifiable rubric with an explicit evidence type and inspection plan. To reduce evaluation cost, these atomic rubrics are hierarchically synthesized into high-level parent rubrics. An agentic judge then evaluates the agent's final workspace against these parent rubrics, returning a success probability along with codebase references. Keeping rubric generation strictly isolated from candidate rollouts avoids trajectory bias, allowing the same rubrics to evaluate diverse candidate models fairly and consistently.

We thoroughly validate AAR across multi-turn benchmarks (SWE-Together and Arena) and single-turn benchmarks (R2E-Gym, SWE-Gym, and SWE-bench Verified). We compare AAR against direct LLM judges, a preference-based checklist baseline, and state-of-the-art rubric verifiers including AgenticRubrics (Raghavendra et al., 2026a) and AdaRubric (Ding, 2026).

In summary, our key contributions are:

- **The AAR Framework:** We introduce a candidate-agnostic verification pipeline that compiles multi-turn coding interactions into verifiable rubrics. By isolating rubric creation from candidate rollouts, AAR eliminates trajectory bias and handles dynamic requirement evolution.
- **Robust Multi-Turn Performance:** Across five LLM models, AAR outperforms existing verifiers on multi-turn benchmarks while remaining competitive on single-turn tasks. Crucially, AAR maintains stable accuracy as session length increases, whereas baseline performance degrades.
- **Hierarchical Rubric Design:** We show that a concise set of parent rubrics is sufficient for effective agentic evaluation, while the fine-grained atomic rubrics provide explainability for failure analysis and error localization.

2 Related Work

LLM judges and rubrics. Using an LLM to evaluate generated output has become standard practice, whether through direct scoring or prompt-embedded rubrics (Zheng et al., 2023; Liu et al., 2023; Kim et al., 2024), and such judges prefer longer answers, their own style, and the user's view (Dubois et al., 2024; Panickssery et al., 2024; Sharma et al., 2024). In a multi-turn coding session, the last bias matters most, since the user's reactions are easier to read than the code. Per-prompt rubrics make judges more reliable and now serve as rewards for reinforcement learning (Viswanathan et al., 2025; Gunjal et al., 2026; Huang et al., 2025), but most automatic rubric generators derive the rubric from a preferred and a rejected response (Liu et al., 2026; Wang & Xiong, 2025; Xie et al., 2025). Our ReqChecklist baseline ports this family to our scenario. The closest verifiers to ours are AgenticRubrics (Raghavendra et al., 2026a), which write rubrics from the issue while exploring the repository and judges the patch diff against it, and AdaRubric (Ding, 2026), which derives task-specific rubrics from the task description and scores the trajectory on them. AAR's rubrics come from the user's turns alone, track requirements that change across turns, and are verified in the final workspace; we compare our method with these baselines in the experiments.

Verifying coding agents. Execution is the natural ground truth where tests exist (Jimenez et al., 2024; Chowdhury et al., 2024; Pan et al., 2025; Jain et al., 2025), but verifiers that do not run code are still needed for test-time selection and as training rewards (Cobbe et al., 2021; Brown et al., 2024; Snell et al., 2025), whether trained on labeled rollouts (Pan et al., 2025; Jain et al., 2025) or prompted as critics (Zhang et al., 2025; McAleese et al., 2024). Agent-as-a-Judge reads the generated project against human-written requirements (Zhuge et al., 2025). Our method reads it against requirements automatically compiled from the session, and needs neither annotators nor training data.

Multi-turn evaluation. Models do worse when instructions arrive over several turns than all at once (Laban et al., 2026; Bai et al., 2024; Wang et al., 2024), and coding requests are often underspecified at first (Vijayvargiya et al., 2026). Multi-turn benchmarks replay or simulate the user and score the outcome with tests or fixed rubrics (Deshpande et al., 2025; Wu et al., 2026; Raghavendra et al., 2026b; King & Flanigan, 2026; Kim et al., 2025). However, none of them provides a verifier that tracks how the requirements change during a session, which is what AAR adds.

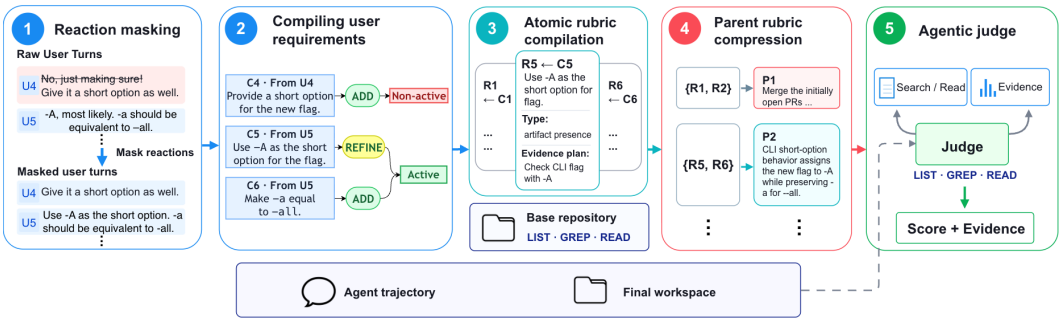

Figure 1: **Agentic Auto-Rubrics (AAR pipeline overview). Steps 1–4 (Compilation):** AAR extracts active user requirements from masked conversation turns, compiles them into atomic rubrics paired with evidence plans using the base repository, and compresses them into a concise set of parent rubrics—without accessing agent responses, rollout workspaces, ground-truth labels, or reference solutions. **Step 5 (Verification):** An agentic judge uses read-only workspace tools to verify the parent rubrics against the final workspace W, returning a success probability alongside concrete verification evidence.

3 METHOD

Problem setup. We formulate the task of verification for multi-turn coding sessions. Given an interactive session with user turns $u_1, \ldots, u_T$ and an agent's final workspace W (the static repository state after turn T), an evaluator $E(u_1, \ldots, u_T, W) \in [0, 1]$ estimates the probability that the agent satisfied the user's overall intent, represented by a hidden ground-truth label $y \in \{0, 1\}$. Instead of running code or executing test suites, E evaluates task completion solely through static inspection of the final repository files in W. This execution-free evaluation is essential for offline benchmark assessment, test-time candidate filtering, and reward modeling during agent training.

To evaluate trajectories systematically, E compiles user turns into a set of criteria called rubrics, denoted as $R = \{r_1, r_2, \ldots, r_K\}$. Each rubric r_k represents a single condition that can be confirmed from file evidence in W. An ideal rubric set R must be *candidate-blind* (derived strictly from user turns without inspecting candidate rollouts), *current* (retaining only active user intent), and *verifiable* (checkable by explicit file paths, line ranges, or code quotes).

We define *active requirements* C_{active} as the instructions still enforced by the user at the end of the session. Because user requests evolve across turns, relying solely on the initial prompt u_1 leaves requirements underspecified, while taking a simple union of all turns mistakenly retains requirements that were later revised. For example, in the task shown in Table 1, the opening turn asks only to merge pull requests, but five additional requirements arrive in later turns, including one that updates a request for an unspecified short flag to the explicit `-A` option.

Overview. As illustrated in Figure 1, AAR operates in two primary phases: a *requirement compilation phase* (Steps 1–4) that turns multi-turn user requests into evaluation rubrics, and an *agentic verification phase* (Step 5) that checks the final workspace W against those rubrics. Full prompt templates, execution parameters, and a trace of an example task appear in Appendix C, Appendix B, and Table 1.

3.1 STEP 1: REACTION MASKING

In multi-turn interactions, users often mix functional requirements with subjective feedback on the agent's previous actions. Evaluators exposed to conversational cues like "perfect, thanks" or "it still crashes" risk guessing task completion rather than verifying code correctness. To eliminate this shortcut, a reaction masking module rewrites each user turn u_t to preserve functional instructions. For example, a feedback turn such as "the button still does nothing" is transformed into a neutral requirement: "The button must open the target dialog." To ensure fair comparisons, all experiments (including baselines) receive masked user turns and are explicitly notified that reactions have been removed.

turn	raw user text	masked rewrite
0	Merge all open PRs (use the `gh` tool) and clean up the feature branches.	Merge all open PRs using the `gh` tool and clean up the feature branches.
1–2	Goodbye. / quit	*no requirement*
3	Okay, now, that new command line flag? Document it in the `-help` text.	Document the new command line flag in the `-help` text.
4	No, just making sure! Give it a short option as well.	Give it a short option as well.
5	-A, most likely. -a should be equivalent to –all.	Use `-A` as the short option. `-a` should be equivalent to `-all`.
6	Commit and push when you are done.	Commit and push when you are done.
7	WHEN YOU ARE DONE	Only when you are done.
8	Not until then	Not until then.
9	Were you done working?	*no requirement*
10	No, just making sure. It seemed like my instruction interrupted your work and you prematurely committed. Sorry.	Commit only when the work is done.
11	Nothing, we are done.	*no requirement*

Table 1: **One task, as the verifier sees it.** The twelve user turns of a SWE-Together task (a Python command-line process monitor to which the user wants a new flag added, plus repository housekeeping) and the masker's rewrite of each. Turn 5 refines turn 4; turns 7, 8 and 10 react to a premature commit and become one neutral requirement. Contract and rubrics: Appendix H.

3.2 Step 2: Compiling user requirements

To track requirement drift across multi-turn sessions, a requirements-analyst prompt processes the masked turns sequentially to construct the *Active contract*. Each extracted specification is tagged as newly introduced (`ADD`), refining a prior requirement (`REFINE`), or superseding an earlier instruction (`SUPERSEDE`), with refined or superseded entries marked inactive. A lightweight QA pass then compares the contract against the source transcript, pruning hallucinated entries or instructions that contradict user intent. In Table 1, a turn-4 request for a short flag is refined at turn 5 to the explicit `-A` flag, deactivating the earlier entry.

3.3 Step 3: Atomic rubric compilation

While a requirement states *what* must be achieved, it rarely specifies *how* an automated evaluator should check it. The atomic rubric compiler expands each active requirement into one or more *atomic rubrics*, each representing a single verifiable condition. Each atomic rubric carries an explicit evidence type indicating what observation would settle it (e.g., code presence, static functionality, runtime behavior, trajectory history, or external repository state) along with an importance level (*critical*, *important*, or *minor*). To ensure precise code verification, the compiler explores the original (base) repository using read-only tools. It locates where the relevant code resides and generates an *evidence plan* that directs the judge to specific files during inspection.

3.4 Step 4: Parent rubric compression

To keep evaluation efficient, AAR compresses these atomic rubrics into at most 10 high-level *parent rubrics*. Every atomic rubric is mapped to a parent rubric without dropping or altering any requirements. A validator LLM then checks that all critical rubrics are fully covered, triggering a one-time retry if any gap is found. Only these parent rubrics are sent to the agentic judge, while the atomic rubrics are saved for detailed diagnostics.

3.5 Step 5: Agentic judge

The final evaluation is performed by an agentic judge that receives the parent rubrics and the masked trajectory. Because trajectories merely record what the agent *claimed* to do, the judge directly inspects the agent's *final workspace* W using read-only tools (`list`, `find`, `grep`, `read`, and `diff` against the base commit). The judge is prompted to verify code and treat source files as data. After at most 15 tool calls, the judge outputs a single *success probability* (estimating $y = 1$) along with supporting evidence: file paths, line ranges, relevant code quotes, and their corresponding rubrics. Rubrics that cannot be verified from W are explicitly recorded as *unresolved rubrics*.

Table 2: **How the verifiers differ**: what each method writes its rubrics from, and how the judge of the main tables reaches a score. "Repo" is the base repository, available where a base commit exists (not on Arena). The methods' own judges, over the patch diff only (AgenticRubrics) or the trajectory only (AdaRubric), are the starred rows of Table 4.

	rubric generation		judge in the main tables	
method	written from	tracks turn changes	sees	score
HOLISTIC	–	–	trajectory	one probability
CONTRACT	user turns → contract	yes	trajectory + contract	one probability
REQCHECKLIST	user turns	no	trajectory + workspace	avg rubric score
AgenticRubrics	user turns + repo	no	trajectory + workspace	weighted sum
AdaRubric	user turns	no	trajectory + workspace	avg rubric score
AAR (ours)	user turns → contract + repo	yes	trajectory + workspace	one probability

Table 3: **Benchmarks**: Arena, real multi-turn sessions from an LLM arena platform with one rollout each; SWE-Together (Wu et al., 2026), scripted multi-turn replays in real repositories; and the single-turn R2E-Gym (Jain et al., 2025), SWE-Gym (Pan et al., 2025) and SWE-bench Verified (Jimenez et al., 2024; Chowdhury et al., 2024). Arena sessions start from the user's own files, so there is no base commit.

			what the judge sees			
dataset	avg. turns	label $y = 1$ when	trajectory	artifact	base commit	rollouts / tasks
Arena	6.6	user's request accomplished	turns + replies	workspace	no	188 / 188
SWE-Together	6.7	test reward $\geq .85$	turns + actions	workspace	yes	264 / 104
R2E-Gym	1	hidden tests pass	issue + actions	patch	yes	1,289 / 363
SWE-Gym	1	hidden tests pass	issue + actions	patch	yes	437 / 120
SWE-bench Verified	1	hidden tests pass	issue + actions	patch	yes	474 / 99

3.6 INFORMATION CHANNELS ACROSS METHODS

Table 2 summarizes the differences between AAR and the baseline methods described in Section 4.1. In our main experiments, all rubric-based methods are evaluated using the same agentic workspace judge (Section 3.5) to isolate the effect of rubric quality. Methods differ primarily in their rubric derivation sources, whether they account for requirement drift across turns, and how the final judge formulates its score. Among all methods, only AAR dynamically tracks requirement changes across turns. AAR-ATOMIC serves as a diagnostic variant where the judge evaluates the complete set of uncompressed atomic rubrics.

4 EXPERIMENTS

4.1 SETUP

Datasets and Ground Truth. We evaluate AAR on two multi-turn benchmarks (Arena, SWE-Together) and three single-turn benchmarks (R2E-Gym, SWE-Gym, SWE-bench Verified), as summarized in Table 3 and Appendix A. Each benchmark trajectory carries a binary ground-truth label $y \in \{0, 1\}$ hidden from all verifiers. On single-turn tasks (R2E-Gym, SWE-Gym, SWE-bench Verified), $y = 1$ indicates that the agent's patch passes the repository's hidden test suite. On SWE-Together, $y = 1$ holds when the final workspace achieves a test-runner reward of at least 0.85. For Arena, which lacks execution tests, y records human-adjudicated task completion based on the final workspace state.

Metrics. Evaluators score each rollout with a success probability, which we evaluate using two standard ranking metrics. *AUROC* assesses global evaluation performance across all benchmark

Table 4: **AUROC with the GPT-5.5 judge** (higher is better). HOLISTIC and CONTRACT read the trajectory only; every rubric method is judged by the same agentic workspace judge. MT avg and ST avg average the multi-turn and the single-turn benchmarks. Bold marks the best of the six ranked methods per column; the gray row is an ablation and is not ranked. *: the method's own judge (patch diff only for AgenticRubrics, trajectory only for AdaRubric). †: Arena sessions with a stored first snapshot as base state (Appendix A).

	multi-turn			single-turn			
method	Arena	SWE-Together	MT avg	R2E-Gym	SWE-Gym	SWE-bench	ST avg
HOLISTIC	.732	.698	.715	.819	.872	.763	.818
CONTRACT	.759	.723	.741	.812	.870	.741	.808
REQCHECKLIST	.799	.718	.759	.796	.883	.709	.796
AgenticRubrics	.739	**.777**	.758	.835	.879	.721	.812
AdaRubric	.763	.750	.757	.801	**.901**	.743	.815
AAR (ours)	**.835**	.763	**.799**	**.850**	.881	**.772**	**.834**
AAR-ATOMIC (ablation)	.832	.766	.799	.857	.877	.778	.837
AgenticRubrics*	.640†	.725	.683	.811	.878	.700	.796
AdaRubric*	.561	.631	.596	.692	.776	.597	.688

instances, measuring the probability that a successful rollout ($y = 1$) scores higher than an unsuccessful one ($y = 0$). Because practical deployment often requires selecting the best rollout for a given task, we also report *Pairwise Accuracy*: the proportion of same-task rollout pairs where the higher-scoring candidate achieves a better outcome. For single-turn benchmarks, the better outcome is defined as $y = 1$; for SWE-Together, it requires a test-reward advantage of at least 0.05. Arena features a single rollout per task and is thus evaluated solely via AUROC.

Baselines. HOLISTIC is the direct scoring judge (Zheng et al., 2023): it reads the masked trajectory and returns a success probability in one call, with no rubric and no tools. CONTRACT is the same judge with our compiled requirement added to its prompt. REQCHECKLIST, a per-requirement checklist ported from preference-mode auto-rubric methods (Wang & Xiong, 2025; Gunjal et al., 2026; Viswanathan et al., 2025), extracts a flat list of requirements from the masked request. AgenticRubrics (Raghavendra et al., 2026a) joins all masked user turns into one task description and writes a checklist from it; where a base repository exists, it explores the repository while writing. AdaRubric (Ding, 2026) writes task-specific scoring dimensions from the same task description. REQCHECKLIST, AgenticRubrics and AdaRubric are judged by the same agentic judge as AAR, and each turns the judge's answers into a score in its own way. The starred rows of Table 4 and Table 5 report instead the published methods' own judges, re-implemented from the papers with GPT-5.5 as the judge: AgenticRubrics's reads the final patch diff only, and AdaRubric's the trajectory step by step, without opening the workspace. Arena has no base commit, so AAR and AgenticRubrics write their Arena rubrics from the user turns alone. AgenticRubrics's own judge needs a repository and a patch, so for its starred row we use the first stored snapshot, the workspace after the agent's first turn.

4.2 MAIN RESULTS

Multi-turn benchmarks. AAR has the highest AUROC on Arena and on the multi-turn average (Table 4): on Arena it reads .835 against .799, .739 and .763 for REQCHECKLIST, AgenticRubrics and AdaRubric under the same judge and .732 for the holistic judge, and of these differences only the one to AgenticRubrics is resolved. On SWE-Together the shared-judge baselines are level with AAR (.777 for AgenticRubrics, .763 for AAR, .750 for AdaRubric), and none of these differences are resolved. The starred rows trail their shared-judge counterparts on both benchmarks, by the most on Arena (Section 5).

Single-turn benchmarks. Here the task is fixed and the requirements do not evolve, so the claim is only that nothing is lost. The four shared-judge methods lie within .07 of one another on each benchmark; AAR has the best single-turn average and the highest point estimate on R2E-Gym and

Table 5: **Within-task selection.** Pairwise accuracy over pairs of rollouts of the same task with different outcomes. Arena has one rollout per task and is not included. Bold and gray as in Table 4; *: the method's own judge.

method	SWE-Together	R2E-Gym	SWE-Gym	SWE-bench
HOLISTIC	.628	.830	.881	**.778**
CONTRACT	.713	.835	.881	.740
REQCHECKLIST	**.716**	.830	.921	.745
AgenticRubrics	.701	.852	.921	.701
AdaRubric	.707	.839	.920	.772
AAR (ours)	.713	**.860**	**.924**	.775
AAR-ATOMIC (ablation)	.713	.864	.929	.775
AgenticRubrics*	.726	.821	.928	.705
AdaRubric*	.652	.738	.819	.592

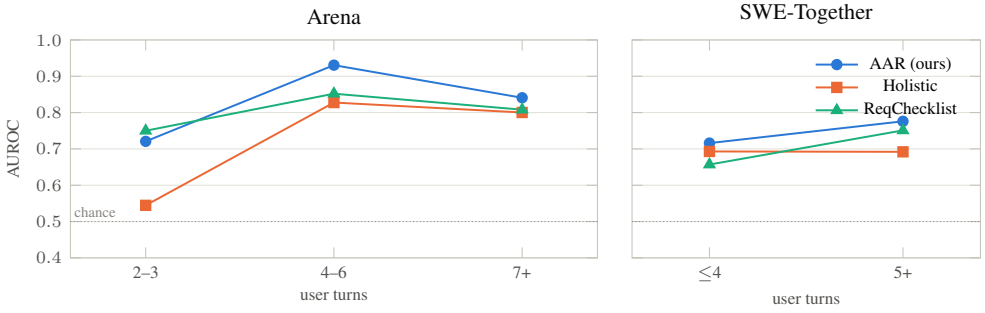

Figure 2: **AUROC by number of user turns.**

SWE-bench Verified, AdaRubric the highest on SWE-Gym (.901 against .881), and no baseline under the shared judge is resolved above AAR in AUROC anywhere. Within tasks (Table 5) AAR is the best selector on R2E-Gym and SWE-Gym, the holistic judge on SWE-bench Verified (.778 against .775).

4.3 PERFORMANCE VERSUS USER TURNS

Figure 2 splits both multi-turn benchmarks by the number of user turns, since verification could be expected to get harder as turns accumulate (Laban et al., 2026). AAR reads .721, .931 and .841 across the three Arena bins and .716 and .776 on SWE-Together, so it does not lose ground with turns; it is ahead of the holistic judge in every bin and ahead of the checklist baseline in four of the five, behind it only on the shortest Arena sessions (.721 against .750). Every bin, with AgenticRubrics and AdaRubric under the same judge, is given in Appendix E.

5 DISCUSSION

The judge matters more than the checklist. Under one shared judge, the flat checklist, AgenticRubrics's checklist, AdaRubric's dimensions and AAR's parent rubrics lie within .07 of one another on SWE-Together and the single-turn benchmarks (Table 4). Scored by their own judges instead, the published methods fall well behind: AdaRubric's judge reads the trajectory step by step and drops from .763 to .561 on Arena, although the holistic judge reads the same trajectory and reaches .732, and AgenticRubrics's judge sees only a patch diff and drops to .640 on Arena, where a session has neither a base commit nor a patch. A verifier that opens the final workspace needs neither.

Knowing what to check and being able to check it are complementary. The CONTRACT judge receives the compiled requirements that AAR is built from but cannot open the workspace; on Arena

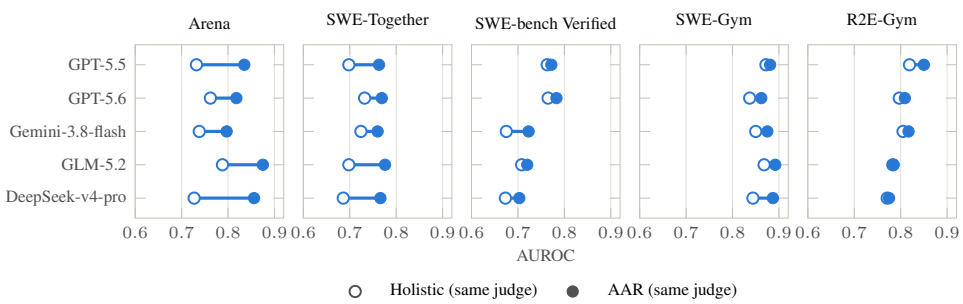

Figure 3: **AAR versus the holistic judge on the same judge model, for five judge models.** Hollow: the holistic judge; filled: AAR; blue where AAR is higher, red where it is lower.

it reads .759, between the holistic judge (.732) and AAR (.835). The requirements alone help a little, the inspection alone helps more (the flat checklist under the agentic judge reads .799), and the two together help most.

Abstraction, not decomposition. Finer rubrics do not make a better verifier. AAR-ATOMIC matches AAR within .013 on every benchmark, but the judge leaves 6.0 atomic rubrics per session unresolved on Arena and 6.4 on SWE-Together, against 0.6 and 2.1 parent rubrics, because narrow rubrics often cannot be settled by reading code and every open rubric weighs on the final decision. Neither a machine-written inspection plan next to the parent rubrics nor a larger tool budget helps (Appendix F).

Verifiable evidence receipts. Every AAR verdict is fully auditable, accompanying its judgment with concrete repository references, averaging 7.3 code citations per session on Arena. A mechanical audit confirms high citation fidelity: 91% of cited quotes appear verbatim in the referenced files, and 99.8% match the cited line ranges (Appendix B). In contrast, baseline judges based purely on full trajectories or raw code diffs provide no such verifiable evidence.

Reliability during test disagreement. When AAR disagrees with execution test results, it rarely acts as an isolated outlier. Across the four test-based benchmarks, 70% of these disagreement cases are backed by at least two other LLM verifiers, while AAR stands alone in only 11% of instances (Appendix D). This strong consensus suggests that disagreements often stem from subtle task ambiguities rather than AAR-specific errors. Furthermore, AAR achieves lower Brier scores than the holistic judge on three of the five benchmarks, demonstrating superior overall probability calibration.

The lead is largest where requirements come from real users. Arena is the one benchmark whose requirements were written by real users over real sessions. It is where the requirements move most, since about 8 percent of the compiled requirements refine or supersede an earlier one, against about 6 percent on SWE-Together and none on the single-turn benchmarks; it is where AAR's margin over every baseline is widest (Table 4); and it is where its advantage does not shrink as turns accumulate (Figure 2).

The conclusions do not depend on the judge model. We reran AAR and the holistic judge with five judge models (Figure 3, Appendix G). With every judge model AAR is at or above the holistic judge on Arena and SWE-Together, and on the single-turn benchmarks its value is higher in every cell that was run.

6 CONCLUSION

We presented AAR, a verifier for multi-turn coding agents that compiles the user's turns, with reactions masked, into candidate-blind rubrics and verifies them against the final workspace with an agentic judge that cites its evidence. Under one shared judge it is the strongest verifier on real multi-turn sessions and on average, and no baseline is measurably ahead of it elsewhere. The returned

evidence could support failure analysis, and multi-turn verifiers could serve as rewards for multi-turn reinforcement learning.

AI USE STATEMENT

Large language models are the object of study in this paper. Every verifier we evaluate, ours and the baselines, is built on commercial or open-weight LLM APIs. We also used AI coding assistants to help write and refactor experiment code and analysis scripts and to help with the wording of this manuscript. The authors reviewed and tested all generated code, recomputed every number in the paper from the per-rollout outputs, and reviewed and edited all prose. No research idea presented as a contribution and no experimental result was produced by AI. We take full responsibility for the content of this paper.

ETHICS STATEMENT

The Arena sessions are real multi-turn coding sessions collected on the Arena.ai platform under its terms of service. Personal identifiers are removed before processing, no user text is quoted in this paper, and per-session workspaces, judge logs and model caches are not released. The other four datasets are built from public open-source repositories and publicly released benchmark instances. Our verifier judges the behavior of coding agents, not of people. Like any automatic verifier, it could be gamed by an agent optimized against it, which is a risk to keep in mind when it is used as a training reward.

REPRODUCIBILITY STATEMENT

Every rubric method in the main tables, REQCHECKLIST, AgenticRubrics, AdaRubric and AAR, was judged by the same judge model at the same reasoning effort, with the same read-only tools and the same tool budget, on the same masked turns and the same trajectories of each dataset. The starred rows are the published methods' own judges and are marked as such wherever they appear. All prompts and output formats are reproduced in Appendix C, the pipeline constants in Appendix B and dataset construction and labels in Appendix A. Code, configuration files, the table builder and per-rollout scores for the public benchmarks will be released.

REFERENCES

Ge Bai, Jie Liu, Xingyuan Bu, Yancheng He, Jiaheng Liu, Zhanhui Zhou, Zhuoran Lin, Wenbo Su, Tiezheng Ge, Bo Zheng, and Wanli Ouyang. MT-Bench-101: A Fine-Grained benchmark for evaluating large language models in Multi-Turn dialogues. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL)*, pp. 7421–7454, Bangkok, Thailand, 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.401.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI blog, `https://openai.com/index/introducing-swe-bench-verified/`, August 2024. Published August 13, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Kaustubh Deshpande, Ved Sirdeshmukh, Johannes Baptist Mols, Lifeng Jin, Ed-Yeremai Hernandez-Cardona, Dean Lee, Jeremy Kritz, Willow E. Primack, Summer Yue, and Chen Xing. MultiChallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier LLMs. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 18632–18702, Vienna, Austria, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.findings-acl.958. arXiv version (2501.17399) lists Sirdeshmukh as first author.

Liang Ding. AdaRubric: Task-Adaptive rubrics for reliable LLM agent evaluation and reward learning. *arXiv preprint arXiv:2603.21362*, 2026. Also presented at the KnowFM workshop at ACL 2026.

Yann Dubois, Balázs Galambosi, Percy Liang, and Tatsunori B. Hashimoto. Length-Controlled AlpacaEval: A simple way to debias automatic evaluators. In *Conference on Language Modeling (COLM)*, 2024.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. In *International Conference on Learning Representations (ICLR 2026)*, 2026. URL https://openreview.net/forum?id=c1bTcrDmt4.

Zenan Huang, Yihong Zhuang, Guoshan Lu, Zeyu Qin, Haokai Xu, Tianyu Zhao, Ru Peng, Jiaqi Hu, Zhanming Shen, Xiaomeng Hu, Xijun Gu, Peiyi Tu, Jiaxin Liu, Wenyu Chen, Yuzhuo Fu, Zhiting Fan, Yanmei Gu, Yuanyuan Wang, Zhengkai Yang, Jianguo Li, and Junbo Zhao. Reinforcement learning with rubric anchors. *arXiv preprint arXiv:2508.12790*, 2025.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve Real-World GitHub issues? In *International Conference on Learning Representations (ICLR 2024)*, 2024.

Myeongsoo Kim, Shweta Garg, Baishakhi Ray, Varun Kumar, and Anoop Deoras. CodeAssistBench (CAB): Dataset & benchmarking for multi-turn Chat-Based code assistance. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025), Datasets and Benchmarks Track*, 2025.

Seungone Kim, Jamin Shin, Yejin Cho, Joel Jang, Shayne Longpre, Hwaran Lee, Sangdoo Yun, Seongjin Shin, Sungdong Kim, James Thorne, and Minjoon Seo. Prometheus: Inducing fine-grained evaluation capability in language models. In *International Conference on Learning Representations (ICLR 2024)*, 2024.

Brendan King and Jeffrey Flanigan. Dialogue SWE-Bench: A benchmark for Dialogue-Driven coding agents. *arXiv preprint arXiv:2606.13995*, 2026.

Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. LLMs get lost in Multi-Turn conversation. In *International Conference on Learning Representations (ICLR 2026)*, 2026. URL https://openreview.net/forum?id=VKGTGGcwl6.

Kilian Lieret, Carlos E. Jimenez, et al. mini-SWE-agent: The 100 line AI agent that solves GitHub issues. GitHub repository, https://github.com/SWE-agent/mini-swe-agent, 2025. SWE-agent team (Princeton University and Stanford University).

Tianci Liu, Ran Xu, Tony Yu, Ilgee Hong, Carl Yang, Tuo Zhao, and Haoyu Wang. OpenRubrics: Towards scalable synthetic rubric generation for reward modeling and LLM alignment. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (ACL)*, pp. 17417–17437, San Diego, California, USA, 2026. Association for Computational Linguistics. doi: 10.18653/v1/2026.acl-long.791.

Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. G-Eval: NLG evaluation using GPT-4 with better human alignment. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2511–2522, Singapore, 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.emnlp-main.153.

Nat McAleese, Rai Michael Pokorny, Juan Felipe Ceron Uribe, Evgenia Nitishinskaya, Maja Trebacz, and Jan Leike. LLM critics help catch LLM bugs. *arXiv preprint arXiv:2407.00215*, 2024.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025.

Arjun Panickssery, Samuel R. Bowman, and Shi Feng. LLM evaluators recognize and favor their own generations. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic rubrics as contextual verifiers for SWE agents. *arXiv preprint arXiv:2601.04171*, 2026a.

Mohit Raghavendra, Anisha Gunjal, Aakash Sabharwal, and Yunzhong He. SWE-Interact: Reimagining SWE benchmarks as User-Driven Long-Horizon coding sessions. *arXiv preprint arXiv:2606.30573*, 2026b.

Mrinank Sharma, Meg Tong, Tomasz Korbak, David Duvenaud, Amanda Askell, Samuel R. Bowman, Newton Cheng, Esin Durmus, Zac Hatfield-Dodds, Scott R. Johnston, Shauna Kravec, Timothy Maxwell, Sam McCandlish, Kamal Ndousse, Oliver Rausch, Nicholas Schiefer, Da Yan, Miranda Zhang, and Ethan Perez. Towards understanding sycophancy in language models. In *International Conference on Learning Representations (ICLR 2024)*, 2024.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations (ICLR 2025)*, 2025.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. AmbigSWE: Interactive agents to overcome underspecificity in software engineering. In *International Conference on Learning Representations (ICLR 2026)*, 2026.

Vijay Viswanathan, Yanchao Sun, Shuang Ma, Xiang Kong, Meng Cao, Graham Neubig, and Tongshuang Wu. Checklists are better than reward models for aligning language models. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025)*, 2025.

Tevin Wang and Chenyan Xiong. AutoRule: Reasoning chain-of-thought extracted rule-based rewards improve preference learning. *arXiv preprint arXiv:2506.15651*, 2025.

Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. MINT: Evaluating LLMs in multi-turn interaction with tools and language feedback. In *International Conference on Learning Representations (ICLR 2024)*, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR 2025)*, 2025.

Yifan Wu, Zhuokai Zhao, Songlin Li, Ho Hin Lee, Jiacheng Zhu, Shirley Wu, Tianhe Yu, Serena Li, Lizhu Zhang, Xiangjun Fan, and Shengzhi Li. SWE-Together: Evaluating coding agents in interactive user sessions. *arXiv preprint arXiv:2606.29957*, 2026.

Lipeng Xie, Sen Huang, Zhuo Zhang, Anni Zou, Yunpeng Zhai, Dingchao Ren, Kezun Zhang, Haoyuan Hu, Boyin Liu, Haoran Chen, Zhaoyang Liu, and Bolin Ding. Auto-Rubric: Learning from implicit weights to explicit rubrics for reward modeling. *arXiv preprint arXiv:2510.17314*, 2025. v1 title: "Auto-Rubric: Learning to Extract Generalizable Criteria for Reward Modeling".

Lunjun Zhang, Arian Hosseini, Hritik Bansal, Mehran Kazemi, Aviral Kumar, and Rishabh Agarwal. Generative verifiers: Reward modeling as Next-Token prediction. In *International Conference on Learning Representations (ICLR 2025)*, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. Judging LLM-as-a-Judge with MT-Bench and chatbot arena. In *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*, 2023.

Mingchen Zhuge, Changsheng Zhao, Dylan Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-Judge: Evaluate agents with agents. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 80569–80611. PMLR, 2025.

A Datasets

This appendix describes, for each evaluation dataset, where the sessions or rollouts come from, how the outcome label was obtained, and whether a base commit was available. Table 3 in the main text summarizes.

A.1 Arena

Arena consists of 188 real multi-turn coding sessions collected on an LLM arena platform with an agentic coding mode. The exported traces hold the user turns and assistant replies but not the agent's tool events. We restore the final workspace snapshot rather than replaying the session, and session texts are scrubbed of credentials and e-mail addresses when loaded. Each session carries an outcome label recording whether the agent accomplished what the user asked by the end of the session. Arena has no base commit: sessions start from the user's own files, so the compiler does not explore a base before writing rubrics and the judge has no `diff` tool. The platform also stores a snapshot after every turn in which the agent changed the workspace. AgenticRubrics's own judge needs a repository to explore and a patch to score, so its starred Arena row in Table 4 uses the first stored snapshot, the state after the agent's first turn. In the shared-judge row AgenticRubrics writes its rubrics from the masked user turns alone, on all 188 sessions.

A.2 SWE-Together

SWE-Together (Wu et al., 2026) provides multi-turn coding tasks reconstructed from real interactive sessions over open-source repositories. We generate rollouts with a scripted user that replays the substantive user turns of the original session, after the first, as a fixed agenda to a coding-agent command-line tool running in an isolated sandbox; the policies are three tiers of one commercial model family. The judged set is 264 rollouts over 104 tasks, up to five per task, and includes only rollouts whose final workspace could be retrieved. The label is the harness's execution reward from running the task's tests against the final workspace; a rollout counts as a success when its reward is at least .85, 34 of the 264 rollouts succeed, and the continuous reward is kept for the within-task metrics. Each task's base commit is checked out as a worktree, so the judge has the `diff` tool and the compiler may explore the base repository (Appendix B).

A.3 R2E-Gym

R2E-Gym (Jain et al., 2025) is a single-turn benchmark of repository issues, and the issue text is the single user turn. The judged set is 1,289 rollouts over 363 instances chosen to have both passing and failing rollouts, with rollouts from several policy models in the benchmark's own scaffold, after removing the rollouts whose patch did not apply to the base repository. The label is whether the rollout's patch passes the instance's tests, as recorded by the benchmark, with no model involved. The base is the true parent of the instance's fix commit, checked out as a worktree; the rollout's patch is applied on top to form the final workspace, and the judge's tools, which include `diff`, are confined to the resulting file tree.

A.4 SWE-bench Verified

The pool is a pilot of 800 rollouts from eight policy models in the mini-SWE-agent scaffold (Lieret et al., 2025) on 100 SWE-bench Verified instances (Chowdhury et al., 2024; Jimenez et al., 2024) chosen for mixed outcomes; the problem statement is the single user turn. The agent submits its work as a diff in its last step, and rollouts that never submitted a patch, or whose patch was cut by the agent environment's output truncation, were removed, leaving 474 rollouts over 99 instances. The label is the per-rollout result of the benchmark's tests, zero or one, checked against the public leaderboard before use. The instance's base commit is checked out as a worktree, the final patch is applied on top, and the tool set includes `diff`.

A.5 SWE-Gym

The rollouts are the test split of the SWE-Gym verifier study (Pan et al., 2025), produced by an OpenHands agent (Wang et al., 2025) over ten repositories; the issue text is the single user turn. The trajectory records the agent's edits as inline editor calls and contains no final patch, so we reconstruct the final workspace by replaying those edits into a fresh worktree at the base commit, checking every edit against the file snapshot in the trajectory. Rollouts whose replay diverged from the trajectory were removed, leaving 437 rollouts over 120 instances, and rollouts in which the agent made no edit are judged at the unmodified base. A control on a separate public SWE-Gym release that ships native patches found that judging replayed and native workspaces gives the same AUROC up to noise. The label is the benchmark's `resolved` flag, an execution result from the instance's tests. The base commit is the worktree the edits are replayed into, and the tool set includes `diff`; the replayed workspace contains everything the agent left behind, including scratch and reproduction files.

B Pipeline details

The pipeline first masks the user turns and compiles the requirement contract from them, with a second model checking the contract. It then expands the active requirements into atomic rubrics and compresses those into parent rubrics, which the judge finally uses to score each rollout. Every prompt is reproduced in Appendix C. Contracts are compiled once per rollout from that rollout's masked turns, and atomic and parent rubrics are compiled once per task and shared by every rollout of the task and by every rubric method. Where a task has several rollouts, as on SWE-Together, the compiler reads the masked user turns of the first rollout in frame order; the scripted user turns are identical across rollouts for 70 of the 82 SWE-Together tasks with more than one rollout. On the single-turn datasets the issue text is the whole contract, so the compiler is told that no requirement lifecycle exists, and the per-rollout contract is consumed only by the Contract judge.

B.1 Models, budgets and tools

Every stage uses GPT-5.5 at the provider's default reasoning effort, except the contract quality pass, which uses gpt-5.4-mini at temperature 0. Each model call gets one retry on malformed output. The atomic-rubric compiler may make up to 12 read-only calls over the base repository where one exists. The agentic judge may make at most 15 tool calls over 25 rounds. Each tool result is cut at 6,000 characters, the masked trajectory is middle-truncated at 20,000 characters and the rubric block at 13,000. The judge's read-only tools are `list` (directory listing), `find` (glob or substring match on paths), `grep` (regular expression over the workspace's files), `read` (numbered line ranges, at most 120 lines per call) and, where a base commit exists, `diff` against it (Arena has none). Paths are confined to the workspace root, dependency and build directories are hidden, and every result is scrubbed of credentials. Pairs whose scores lie within .05 count as ties in pairwise accuracy.

Diff channel. On the three single-turn benchmarks an earlier run's `diff` tool returned the diff of the wrong repository. The tool was repaired to return the rollout's own diff against the base commit. Every run reported in the paper's tables used the repaired tool, and the recompiled runs of AAR and AAR-Atomic replaced the earlier ones (Appendix F).

Definitions. Contract requirements are marked ADD, REFINE or SUPERSEDE, and the QA model marks each requirement ok, inverted, wrong-turn or not-asked. The active contract is the set of requirements still active at the end of the session minus those marked not-asked or inverted. Atomic rubrics carry one of five types, routed as in Table 6, and an importance level of critical, important or minor. A parent rubric's importance is the maximum over its children. For n atomic rubrics the parent compressor must return between $\min(5, \max(1, \lceil n/2 \rceil))$ and 10 parent rubrics, and its output is rejected if more than 5% of the critical and important atomic rubrics are left uncovered by any parent rubric. The judge's final answer contains a 0–100 score, a success probability, evidence (file, lines, quote, parent-rubric id), the ids of the parent rubrics it could not settle, and a short rationale.

Table 6: Evidence types of atomic rubrics and the channel each is routed to. The judge may settle a rubric only where marked; everything else is evidence that feeds its overall score or is reported as unresolved.

type	channel	judge may settle
artifact presence	artifact (static)	yes
artifact semantics	artifact (static)	evidence only
execution behaviour	execution	evidence only
trajectory fact	trajectory	yes
external state	unresolved	evidence only

B.2 Sizes of the compiled contracts and rubrics and of the judge's work

Table 7 gives the sizes of the contracts, atomic rubrics and parent rubrics on every dataset, and Table 8 what the judge did with them. Both tables describe the canonical compile, in which base-repository exploration happened only on the 56 SWE-Together tasks whose checkout was materialized at compile time, and the three single-turn datasets compiled their atomic rubrics from the issue text alone. In the runs reported in the main tables the compiler explored a base checkout on every task with a base commit, 103 of the 104 SWE-Together tasks and all single-turn tasks, with 10.8 exploration calls per task on SWE-Together and 7.2 to 7.6 on the single-turn benchmarks (Appendix F); Arena has no base repository.

Table 7: Sizes of the compiled contracts and rubrics in the canonical compile. Contracts are per rollout, and atomic and parent rubrics are per task. Type and importance shares are over all atomic rubrics of the dataset.

	Arena	SWE-Together	R2E-Gym	SWE-bench Verified	SWE-Gym
requirements per contract, mean / median	11.24 / 12.0	11.86 / 13.0	4.39 / 4	4.33 / 4	4.28 / 4
atomic rubrics per task, mean / median / min / max	19.79 / 13.5 / 2 / 101	13.72 / 13.0 / 3 / 31	4.03 / 4 / 1 / 9	4.80 / 4.5 / 2 / 12	4.62 / 4.0 / 1 / 10
compile tool calls per task, mean / max	0.00 / 0	1.59 / 10	0.00 / 0	0.00 / 0	0.00 / 0
artifact presence	46.2%	25.9%	20.7%	23.8%	25.8%
artifact semantics	31.9%	30.1%	36.4%	46.2%	44.7%
execution behaviour	13.9%	9.0%	42.9%	30.0%	29.5%
trajectory fact	7.6%	27.8%	0	0	0
external state	0.4%	7.2%	0	0	0
critical / important / minor	52.4 / 46.2 / 1.3%	47.0 / 48.0 / 5.0%	56.7 / 43.1 / 0.2%	49.0 / 50.4 / 0.6%	51.7 / 48.1 / 0.2%
parent rubrics per task, mean / median / min / max	5.72 / 6.0 / 1 / 10	5.98 / 6.0 / 2 / 9	2.73 / 3 / 1 / 6	3.24 / 3.0 / 1 / 6	3.25 / 3.0 / 1 / 6

Parent rubrics leave about 0.6 unresolved rubrics per rollout on Arena, against about six unresolved atomic rubrics, which is the reason for compressing in the first place. The judge is not starved of budget on R2E-Gym and SWE-bench Verified, where the median run ends below the cap, while on Arena and SWE-Together most rollouts use the whole budget.

Table 8: Agentic-judge statistics per rollout for the two rubric readings in the canonical runs (gpt-5.5, cap of 15 tool calls). "Calls" counts accepted tool calls, and "unresolved" is the number of rubrics the judge reported it could not settle.

dataset	method	calls, mean / median	evidence entries per rollout	unresolved per rollout
Arena	AAR	13.11 / 15	7.35	0.61
	AAR-ATOMIC	13.23 / 15	7.79	5.99
SWE-Together	AAR	13.87 / 15	5.34	2.05
	AAR-ATOMIC	13.87 / 15	6.10	6.40
R2E-Gym	AAR	8.97 / 9	3.24	0.04
	AAR-ATOMIC	9.01 / 9	3.63	1.54
SWE-bench Verified	AAR	11.75 / 12.5	3.95	0.09
	AAR-ATOMIC	11.97 / 13	4.43	1.21
SWE-Gym	AAR	12.85 / 14	3.86	0.10
	AAR-ATOMIC	13.06 / 15	4.26	1.26

B.3 CHECKING THE EVIDENCE

Every judged rollout stores the judge's evidence and tool calls, and a mechanical audit can be run on them afterwards. A piece of evidence counts as verified when its file resolves to a regular file under the workspace root, its quote has at least 8 characters after whitespace runs are collapsed and case is folded, and that normalized quote occurs in the file's text. On Arena the AAR judge with gpt-5.5 returns 7.3 pieces of evidence per session and on SWE-Together 5.4. Of the quoted snippets, 91% on Arena and 95% on SWE-Together are found in the cited file after whitespace and case normalization, 4.9% and 3.4% name a file that does not exist in the workspace, and 99.8% of the found quotes lie within the cited line range. The audit checks existence and containment, not whether the quote supports the rubric it is attached to.

C PROMPTS AND OUTPUT SCHEMAS

This appendix presents the core system and user instruction prompts across all pipeline stages and evaluators in sequential order. Placeholders in angle brackets (e.g., <USER TURNS BLOCK>) indicate runtime content.

C.1 STAGE 1: REACTION MASKER

System Prompt

```
You are a meticulous redaction engine preparing research data. You
rewrite the user's messages from an interactive coding session so that
all EVALUATIVE REACTION content is removed while all REQUIREMENT content
 is kept. You never add information that is not in the original messages
.
```

User Prompt & Task Instruction

```
Rewrite EACH turn, keeping only requirement content:
- KEEP: task requirements, specifications, constraints, parameters,
questions that carry requirements, and factual material supplied as
input (code, data, file names).
- REMOVE: praise, thanks, complaints, satisfaction or dissatisfaction,
acceptance/approval ('looks good'), frustration, and commentary on
whether work was right or wrong.
- Rephrase corrections and problem reports as NEUTRAL requirements in
their final form: state what the delivered work must do, without
indicating that anything was previously wrong.
- Keep the rewrite in the user's voice, concise, faithful; never invent
requirements.
- If nothing remains after removal, use an empty string and
has_requirements=false.

Return ONLY a JSON object:
```

```
{"turns": [{"turn": <int>, "masked": "<rewritten text>", "
has_requirements": true|false}, ...]}
```

C.2 Stage 2: Requirement Contract Compiler & QA

Contract Compiler Instruction

```
Compile the user's requirements into a SPECIFICATION: a list of claims.
Each claim is one atomic, independently checkable requirement about what
 the delivered code must do, stated in its FINAL form.

Rules:
- Strip reactions: Record ONLY requirements. Do NOT encode praise,
complaints, or corrections.
- If a later turn CHANGES a requirement, state only the final
requirement; mark any superseded original inactive.
- Lifecycle ops: op=ADD (new requirement), op=REFINE (sharpens existing
claim), op=SUPERSEDE (replaces old claim). Do NOT use CORRECT, REDIRECT,
 ACCEPT, or VERIFY.
- Claims must come ONLY from what the user wrote. 3 to 16 claims total.

Return ONLY a JSON object:
{"claims": [{"claim_id": "c1", "claim": "...", "introduced_at_turn": <
int>, "op": "ADD|REFINE|SUPERSEDE", "supersedes": ["c0"], "active": true
|false}, ...]}
```

Contract QA Audit Instruction

```
Audit task-contract claims extracted from user turns. For each claim,
assign a verdict:
- ok: faithful to user requests
- inverted: rewards the OPPOSITE of what was asked
- wrong_turn: anchored to an earlier request superseded by a later turn
- not_asked: unsupported by user turns (invented)

Return ONLY a JSON object:
{"criteria": [{"id": "<claim_id>", "verdict": "ok|inverted|wrong_turn|
not_asked", "note": "<max 15 words>"}], "missing": ["<important ask
absent>", ...]}
```

C.3 Stage 3: Atomic Rubric Compiler

System Prompt

```
You compile a multi-turn user interaction into TASK-SPECIFIC, CHANNEL-
AWARE verification rubrics. The final rubric must be CANDIDATE-
INDEPENDENT: derived strictly from user intent and base repo state.

Rules:
- Each rubric carries EXACTLY ONE decisive claim (split compound
requirements).
- Classify claim_type: "artifact_presence", "artifact_semantics", "
execution_behavior", "trajectory_fact", "external_state".
- Specify materiality ("critical" | "important" | "minor") and
unresolved_conditions.
```

Output Schema & Protocol

```
Return ONLY a JSON object:
{
  "task_summary": "<2-4 sentences, self-contained>",
  "rubrics": [
    {"id": "r1", "claim_ids": ["c1"], "semantics": "<the ONE thing that
must hold>", "claim_type": "...", "absence_claim": false, "materiality":
 "critical|important|minor", "unresolved_conditions": []}
  ],
```

```
  "evidence_plan": {
    "search_concepts": ["<term>"],
    "likely_regions": [{"glob": "<path>", "symbols": ["<names>"], "
confidence": 0.0-1.0, "rubric_ids": ["r1"]}],
    "preferred_checks": [{"cmd": "<command>", "kind": "test|user|build",
 "expect": "...", "rubric_ids": ["r1"]}],
    "fallback": "<search instruction>"
  },
  "exclusions": ["<out-of-scope / superseded>"]
}
```

C.4 Stage 4: Parent Rubric Compressor

Compressor Instruction

```
Reorganize fine-grained atomic rubrics into a small set of PARENT
CRITERIA (5-10 parents). Preserve the full meaning of the atomic set
without inventing new requirements. Each parent criterion must be ONE
readable, human-level sentence.

Rules:
- Every critical or important atomic rubric MUST appear in exactly one
parent's children.
- Parent materiality = highest materiality among its children.

Return ONLY a JSON object:
{"parents": [{"id": "P1", "criterion": "<one human-readable sentence>",
"materiality": "critical|important|minor", "children": ["r1"], "plan":
[{"rubric_id": "r1", "check": "<what to verify>", "where": "<file/dir
hint>"}]}], "uncovered_rubric_ids": []}
```

C.5 Stage 5: The AAR Judge

System & Task Instruction

```
You are an expert evaluator of AI coding agents. You judge whether the
agent satisfied the user's requirements using the interaction trajectory
, a set of PARENT CRITERIA, and read-only tools over the final workspace
. Verify, don't assume. Inspection findings bind ONLY to the criterion
they were gathered for. End with ONE holistic judgment.

Return ONLY a JSON object:
{"score": <integer 0-100>, "success_probability": <float 0.0-1.0>, "
evidence_receipts": [{"file": "<path>", "lines": "<line range>", "quote
": "<snippet>", "parent_id": "<parent_id>"}], "unresolved_parent_ids":
[], "rationale": "<max 60 words>"}
```

D Score alignment and disagreements with the tests

AUROC says how well a verifier ranks; this appendix asks two further questions of the released per-rollout scores. First, how do the scores themselves align with the outcome: are the successes scored high and the failures low, and how often is a confident score wrong? Second, when AAR disagrees with the tests, does it disagree alone, or do the other verifiers read the same rollout the same way? On the four repository benchmarks every AAR number in this appendix, the disagreement counts included, comes from the canonical compile, whose AUROC differs from the recompiled run of Table 4 by at most .018 (R2E-Gym; .011 on SWE-Together, .005 on SWE-Gym, .003 on SWE-bench Verified); the Arena numbers are identical in both. A probability at or above .5 counts as a pass, for every verifier.

Score alignment. Figure 4 shows the distribution of the success probability for successful and failed rollouts, for the Holistic judge and AAR on the same judge model, and Table 9 summarizes it with the flat checklist under the same judge added. AAR ranks at or above the holistic judge

on every benchmark (AUROC; only the Arena difference is resolved), and the gap between its mean probability on successes and on failures is wider on four of the five, narrower on SWE-bench Verified (.251 against .275). Its area under the precision–recall curve is at or above the holistic judge's on four of the five benchmarks and .002 below it on SWE-Gym (.906 against .908), most clearly above on Arena (.921 against .869) and SWE-Together (.284 against .217, where 13 percent of rollouts succeed). Taken over both tails, its confident scores are wrong less often on four of the five benchmarks: the two-sided conclusive-error rate at .85, the share of rollouts scored at least .85 or at most .15 whose outcome lies on the other side, is .10 against .20 on Arena, .11 against .13 on R2E-Gym, .11 against .26 on SWE-Gym and .15 against .16 on SWE-bench Verified, and .20 against .15 on SWE-Together. The gain is at the low tail: among rollouts scored at most .15, the share that succeeded is .114 against .317 on SWE-Gym and .070 against .157 on SWE-bench Verified. At the high tail the two are level or the holistic judge is slightly better, and AAR gives a confident pass to more rollouts; on Arena the share of confident passes that failed is .049 against .077 (2 of 41 against 1 of 13). The Brier score, which rewards calibration as well as ranking, is lower for AAR on Arena, R2E-Gym and SWE-Gym, level on SWE-bench Verified and higher on SWE-Together. The flat checklist under the same judge has the higher precision–recall area on SWE-Together (.330) and fewer wrong confident passes on SWE-Together and SWE-Gym; its ranking is worse (Table 4) and its low tail is not better (.273 of its confident fails on Arena succeeded, against .222 for AAR). A possible reason for SWE-Together is its label: a rollout counts as a failure below a reward of .85, so a verifier that credits a partially successful rollout is scored as wrong. We have not checked this against the reward values, which are not part of the released scores; it is a hypothesis for the audit by hand.

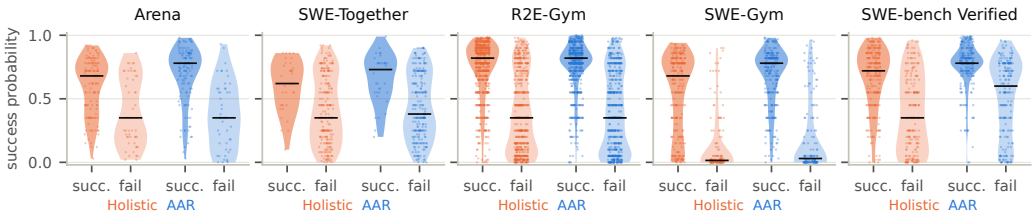

Figure 4: **Success probability by outcome.** For each benchmark, the distribution of the verifier's success probability over successful and failed rollouts, for the HOLISTIC judge (orange) and AAR (blue), both with gpt-5.5; the bar is the median. AAR rows on the four repository benchmarks are the canonical compile.

Where AAR disagrees with the tests. On the four benchmarks scored by tests we take every rollout on which AAR's side of .5 disagrees with the test outcome and ask how the three other verifiers under the same judge model that read the same masked trajectory, the HOLISTIC and CONTRACT judges and the flat checklist under the agentic judge, score it. The published methods' own judges are left out, AdaRubric's because its scores are nearly constant (balanced accuracy .50 at .5 on every benchmark) and AgenticRubrics's because it passes most failing rollouts (.34 of SWE-Together failures scored at least .85); the shared-judge AgenticRubrics and AdaRubric rows are not among the released scores. Table 10 gives the counts. AAR disagrees with the tests on 623 of the 2,464 rollouts. In 437 of them at least two of the three other verifiers take AAR's side against the test, and in 66 AAR stands alone; in the other direction, on 203 rollouts AAR sides with the test while at least two of the other verifiers do not. Most disagreements, 468 of the 623, are rollouts that fail the tests and that AAR passes; on every benchmark but SWE-Gym this is the majority direction, and in 59 to 84 percent of these cases at least two of the three other verifiers pass the rollout too. These are the rollouts a human audit should look at first. Because the four verifiers share the judge model and the masked trajectory, and two of them the compiled requirements, their agreement is not independent evidence that the test is wrong: a shared disagreement rules out an error specific to AAR's rubrics, but is equally consistent with a requirement that reading cannot settle, with a test that asks for something the request did not, and with a misreading common to all four. The disagreements in which AAR stands alone, 66 rollouts in all, are the ones in which an error, if there is one, is AAR's alone. The list of these rollouts, with every verifier's score, is released with the analysis script; the categorization by hand remains to be done.

Table 9: **Score alignment** of three verifiers under the same judge model; on the four repository benchmarks the AAR row is from the original compile (see text). $\bar{p}$ is the mean success probability over successful or failed rollouts; the next two columns are the share of rollouts scored at least .85 that failed and the share scored at most .15 that succeeded, with the number of such rollouts in parentheses; $\text{HCE}_{.85}$ is the share of rollouts scored at least .85 or at most .15 whose outcome lies on the other side. Brier and HCE are given for the two methods whose score is a probability.

dataset	method	AUPRC	$\bar{p}$ succ.	$\bar{p}$ fail	$P(\text{fail} \mid p \geq .85)$	$P(\text{succ.} \mid p \leq .15)$	Brier	$\text{HCE}_{.85}$
Arena	HOLISTIC	.869	.604	.394	.077 (13)	.294 (17)	.209	.200
	REQCHECKLIST	.899	.715	.450	.111 (36)	.273 (11)	–	–
	AAR	.921	.698	.371	.049 (41)	.222 (18)	.156	.102
SWE-Together	HOLISTIC	.217	.599	.416	.692 (13)	.019 (52)	.238	.154
	REQCHECKLIST	.330	.577	.398	.429 (7)	.000 (39)	–	–
	AAR	.284	.658	.439	.706 (17)	.000 (43)	.245	.200
R2E-Gym	HOLISTIC	.811	.743	.380	.158 (348)	.086 (233)	.170	.129
	REQCHECKLIST	.782	.774	.529	.167 (312)	.117 (60)	–	–
	AAR	.811	.762	.392	.156 (289)	.051 (216)	.163	.111
SWE-Gym	HOLISTIC	.908	.558	.127	.078 (51)	.317 (167)	.216	.262
	REQCHECKLIST	.899	.704	.304	.062 (65)	.019 (54)	–	–
	AAR	.906	.682	.191	.103 (58)	.114 (114)	.140	.111
SWE-bench Verified	HOLISTIC	.791	.670	.395	.169 (89)	.157 (70)	.197	.164
	REQCHECKLIST	.728	.725	.561	.205 (73)	.059 (17)	–	–
	AAR	.797	.767	.516	.190 (84)	.070 (43)	.196	.150

Table 10: **Disagreements between AAR and the tests**, a probability at or above .5 counting as a pass, and how the three other verifiers under the same judge model (HOLISTIC, CONTRACT, REQCHECKLIST) score the same rollouts: the cases in which at least two of them take AAR's side, and those in which none does. "Tasks" is the number of distinct tasks the cases fall in; "reverse cases" counts the rollouts with the same test outcome on which AAR agrees with the test while at least two of the other verifiers do not. AAR rows on these benchmarks are the canonical compile.

dataset	AAR vs. the tests	cases	others side with AAR	AAR alone	tasks	reverse cases
SWE-Together	fails a passing rollout	6	3 (.50)	1 (.17)	5	6
	passes a failing rollout	97	57 (.59)	20 (.21)	60	22
R2E-Gym	fails a passing rollout	72	36 (.50)	22 (.31)	51	19
	passes a failing rollout	228	191 (.84)	2 (.01)	146	72
SWE-Gym	fails a passing rollout	57	38 (.67)	11 (.19)	27	47
	passes a failing rollout	26	16 (.62)	1 (.04)	25	2
SWE-bench Verified	fails a passing rollout	20	14 (.70)	2 (.10)	14	22
	passes a failing rollout	117	82 (.70)	7 (.06)	67	13

E TURN ANALYSIS

Figure 2 reads verifier accuracy against the number of user turns; Table 11 gives every bin behind it. No session was re-judged: each bin restricts the scores of the main results to the sessions that fall in it. User turns are counted on the masked trajectory, and every session has at least two. The bins were fixed before any value was computed. On Arena they are near-terciles of the turn count (2–3, 4–6, 7+). SWE-Together has only 34 successful rollouts, so its terciles would hold too few successes to read, and we split it once, at four turns, placing each rollout by its own count of masked user turns. The bins are small, so we read the direction of change across bins rather than any single cell.

Table 11: AUROC by user-turn bin, all with the gpt-5.5 judge. REQCHECKLIST, AgenticRubrics and AdaRubric are the shared-judge rows of the main tables. n is the number of sessions in the bin, with successes and failures in parentheses where they were recorded.

dataset	user turns	n (succ./fail.)	AAR	HOLISTIC	REQCHECKLIST	AgenticRubrics	AdaRubric
Arena	2–3	68 (53/15)	.721	.545	.750	.651	.697
Arena	4–6	62 (44/18)	.931	.828	.852	.794	.795
Arena	7+	58 (39/19)	.841	.800	.808	.771	.761
SWE-Together	≤ 4	101	.716	.693	.657	.737	.676
SWE-Together	5+	163	.776	.692	.751	.808	.803

On Arena every rubric method under the shared judge is lowest on the shortest sessions and none loses ground with turns; on SWE-Together every rubric method rises from the shorter to the longer bin while the HOLISTIC judge stays flat. The bins are small, so this is a direction rather than a measurement.

F ABLATION DETAILS

This appendix gives the numbers behind the ablations of Section 5. Every run uses the main judge, GPT-5.5, with the same prompts, tools, and budgets as the main experiments unless stated otherwise.

F.1 TOOL BUDGET

Design. On a fixed random subsample of 120 sessions from an earlier collection window of the same platform as Arena, disjoint from the 188 sessions of the main text, we re-judged AAR and AAR-ATOMIC with tool-call caps of 8, 25 and 40, reusing the rubrics of the main run unchanged. The cap-15 points are the main-run rows restricted to the same sessions. Everything else is as in the main experiments, except that cap 40 ran with a round limit of 50 rather than 25 so that the budget could actually be spent. The sweep was planned to answer two questions. Does a larger budget let the judge settle more rubrics? And if so, does the atomic reading catch up with the parent reading once it is given more budget? A yes to both would mean that the difference between the two readings is a budget effect rather than an effect of compressing. Table 12 lists the cells.

Table 12: Tool-budget sweep on the 120-session subsample. "Unresolved" is the mean number, and the mean fraction, of the task's rubrics that the judge reported it could not settle. "At cap" is the share of rollouts that used the whole budget.

method	cap	AUROC	unresolved count	unresolved fraction	tool calls	at cap
AAR-ATOMIC	8	.849	9.933	.423	7.70	.900
AAR-ATOMIC	15	.831	8.042	.361	13.25	.725
AAR-ATOMIC	25	.807	6.303	.306	19.12	.445
AAR-ATOMIC	40	.813	5.458	.275	23.84	.167
AAR	8	.834	1.100	.159	7.64	.900
AAR	15	.843	0.583	.087	13.18	.683
AAR	25	.843	0.331	.049	18.52	.407
AAR	40	.794	0.158	.022	22.95	.150

Results. The answer to the first question is yes. The unresolved fraction falls steadily with the cap for both readings and is higher for the atomic reading at every cap, so more budget does let the judge settle more of what it is asked. The answer to the second question is no. AUROC does not improve with the budget for either reading, and the gap between the two readings stays within a few hundredths at every cap without moving in a consistent direction as the budget grows. The difference between parent and atomic rubrics is therefore not a budget effect. The sweep was not designed to compare one cap against another for the same reading, so we do not read the lower point at cap 40 as evidence that a larger budget hurts. The safe conclusion is that caps from 8 to 25 are indistinguishable on this subsample and that the cap of 15 used in the main experiments leaves nothing on the table.

F.2 REACTION MASKING

Every method reads masked user turns on every dataset, so the main experiments contain no masked-versus-unmasked comparison. The evidence that masking is necessary comes from the study that preceded the frozen pipeline, run on an earlier set of sessions from the same platform as Arena. There, letting the HOLISTIC and CONTRACT judges read the unmasked turns raised their AUROC by about $+.08$ to $+.10$ without any change to the agent's work under judgment, which is the signature of a judge reading the user's verdict rather than forming its own. A scripted user that replays fixed requests produces few reactions, and a matched masked-versus-unmasked comparison on SWE-Together changed little. The masker is an LLM and not a guarantee. A few reaction words survive it, and since every method sees the same masked turns, the residue affects them alike. On the single-turn datasets the issue text is passed through unchanged, and the same notice that reactions may have been removed is appended for every method on every dataset.

G JUDGE ROBUSTNESS

Table 13 gives the full matrices behind Figure 3: AAR and the HOLISTIC judge with each of five judge models on the five datasets of the main text. Only the judge changes; the rubrics are those compiled with gpt-5.5 without base exploration, so the gpt-5.5 row differs from Table 4 on the four repository benchmarks (Appendix F). Three cells were not run: gpt-5.6 on R2E-Gym and SWE-Gym, and deepseek-v4-pro on R2E-Gym.

Table 13: AUROC of AAR (top) and of the HOLISTIC judge (bottom) for five judge models.

judge	Arena	SWE-Together	R2E-Gym	SWE-Gym	SWE-bench Verified
AAR					
gpt-5.5	.835	.763	.850	.881	.772
gpt-5.6	.818	.769	.809	.862	.783
gemini-3.8-flash	.797	.760	.817	.875	.723
glm-5.2	.875	.776	.785	.892	.720
deepseek-v4-pro	.856	.766	.775	.887	.703
HOLISTIC *judge*					
gpt-5.5	.732	.698	.819	.872	.762
gpt-5.6	.762	.732	.797	.837	.765
gemini-3.8-flash	.738	.724	.805	.850	.675
glm-5.2	.788	.698	.783	.868	.708
deepseek-v4-pro	.727	.686	.771	.844	.673

Reading the matrices. For every one of the five judges, AAR is at or above the HOLISTIC judge with the same model on Arena and on SWE-Together. On the three single-turn benchmarks AAR's value is higher than the HOLISTIC judge's in every cell that was run. The open-weight judges glm-5.2 and deepseek-v4-pro give the highest Arena values of all, .875 and .856.

H WORKED EXAMPLE

This appendix follows one SWE-Together task, a small Python command-line process monitor to which the user wants a new "rescan for new processes" flag added together with some repository housekeeping, through the compile side of the pipeline.

The contract compiler (Section 3.2) returns the requirements in Table 14, where turn 4's unspecified short option is refined at turn 5 into the explicit `-A`, so `c4` becomes inactive.

Table 14: The requirement contract of the example task.

id	turn	op	supersedes	active	requirement
c1	0	ADD	–	true	Merge all open pull requests using the `gh` tool.
c2	0	ADD	–	true	Clean up the feature branches after merging the open pull requests.
c3	3	ADD	–	true	Document the new command-line flag in the `-help` text.
c4	4	ADD	–	**false**	Provide a short option for the new command-line flag.
c5	5	REFINE	c4	true	Use `-A` as the short option for the new command-line flag.
c6	5	ADD	–	true	Make `-a` equivalent to `-all`.
c7	6	ADD	–	true	Commit the changes only after all requested work is complete.
c8	6	ADD	–	true	Push the changes only after all requested work is complete.

The atomic compiler (Section 3.3) explores the base repository and writes the atomic rubrics in Table 15.

Table 15: Atomic rubrics of the example task.

id	from	type / importance	rubric
r1	c1	external state / critical	All pull requests that were open for the repository at the time of the initial request must have been merged using the GitHub CLI.
r2	c2	external state / important	Feature branches associated with the merged pull requests must have been cleaned up after those pull requests were merged.
r3	c3	artifact presence / important	The program's `-help` output must document the new command-line flag.
r4	c5	artifact presence / critical	The new command-line flag must be invocable with `-A` as its short option.
r5	c6	artifact semantics / critical	The `-a` short option must be equivalent to the existing `-all` option.
r6	c7	trajectory fact / important	Any commit made by the agent must have occurred only after all requested work was complete.
r7	c8	external state / important	Any push made by the agent must have occurred only after all requested work was complete.

The compressor (Section 3.4) returns four parent rubrics that cover all seven atomic rubrics (Table 16), and these four sentences are all the judge receives as its rubric.

Table 16: Parent rubrics of the example task.

id	parent rubric	importance	children
P1	The GitHub pull-request workflow is completed by merging the initially open PRs with gh and removing their feature branches.	critical	r1, r2
P2	The CLI help output documents the new command-line flag.	important	r3
P3	CLI short-option behavior assigns the new flag to -A while preserving -a for –all.	critical	r4, r5
P4	Commit and push actions occur only after all requested repository and external workflow work is complete.	important	r6, r7