SPECIALJUDGEBENCH: VERDICT PREDICTION AND EXECUTABLE JUDGE REPOSITORY SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

Competitive programming benchmarks typically evaluate programs generated by large language models (LLMs) using existing judges. Yet it remains unclear whether LLMs can perform the judging itself, especially for constructive, interactive, and communication problems whose correctness depends on constraints, protocols, and information isolation rather than output matching. To study this gap, we introduce SpecialJudgeBench, comprising 270 problems and 4,540 candidate programs, 903 of which have numerical reference scores. The benchmark assesses two complementary capabilities: predicting reference verdicts and scores for candidate programs, and synthesizing complete executable judge repositories from public problem specifications. We evaluate generated repositories on hidden candidate programs through binary verdict agreement, alongside build, protocol, and runtime checks. Across seven models, even the best verdict accuracy stays below 90% for each problem type; all models have higher false-rejection than false-acceptance rates, and their valid score predictions underestimate reference scores on average. Across eight model–scaffold configurations, execution completion reaches 90.9%–98.8%, but only 42.2%–55.9% of repositories correctly judge every hidden candidate in their evaluation pools. Whether generative models directly judge candidate programs or are tasked with building reliable, complex judging systems, achieving trustworthy evaluation remains a substantial challenge.

1 INTRODUCTION

Reliable program evaluation underpins model comparison, training-data selection, and verifiable rewards for large language models (LLMs); false acceptance, false rejection, and scoring errors compromise these signals (Liu et al., 2023; Zheng et al., 2026). Competitive programming offers explicit specifications and executable feedback, yet benchmarks such as CodeContests (Li et al., 2022), LiveCodeBench (Jain et al., 2024), and LiveOIBench (Zou et al., 2025) primarily follow a *solver-centric* paradigm: models generate contestant programs evaluated by an existing judge. This leaves a central question unanswered: can LLMs reliably judge candidate programs and construct complete judging systems? Answering this question informs the use of LLMs in contest problem preparation, validation, and the development of dependable evaluation infrastructure.

We study three major settings for special judging: constructive, interactive, and communication problems (Figure 1). Constructive problems admit multiple valid outputs, requiring constraint or quality checks. Interactive problems require programs to acquire hidden information through a bounded protocol (Li et al., 2026). Communication problems involve cooperating components with private inputs, requiring information separation and communication limits. All three settings can award partial credit based on subtasks, solution quality, or resource costs (Maggiolo & Mascellani, 2012; Zou et al., 2025). We use *special judging* to encompass their checkers, interactors, communication graders, scoring rules, and execution environments, which evaluate behavior beyond static answer matching.

These settings combine *specification reasoning* with *repository-level engineering*. Correct checking logic can still produce wrong binary verdicts when paired with faulty process coordination, information isolation, or protocol enforcement. We therefore use complete executable repositories as the unit of generation. Within a prescribed environment and using prescribed interfaces, models provide build configurations, entry points, and judging components that manage candidate compilation and

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.

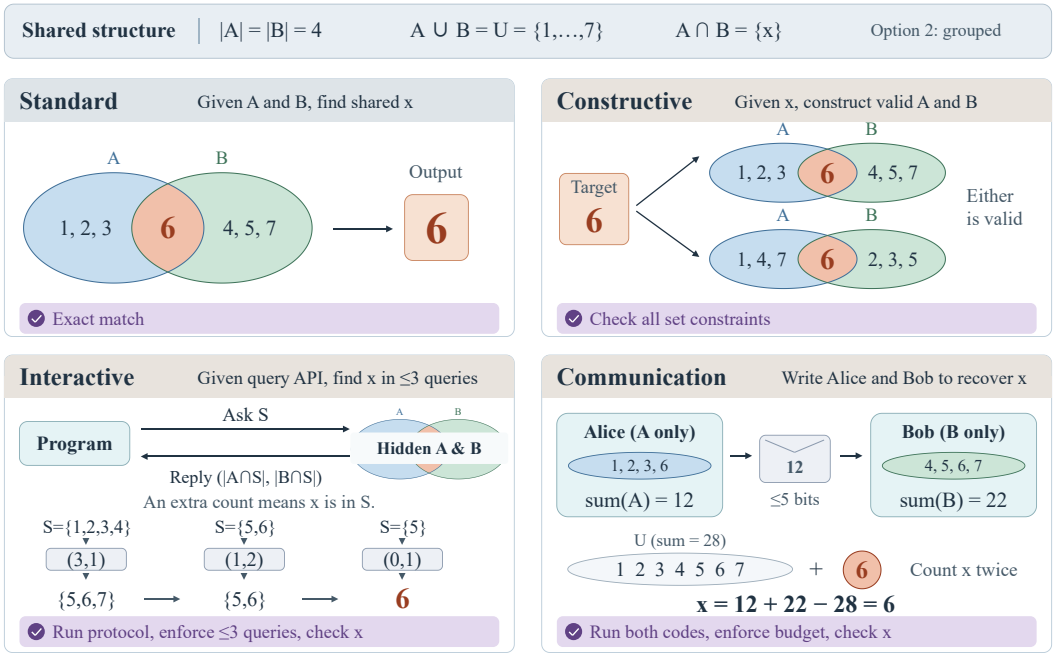

Figure 1: Four judging settings illustrated by a shared set-recovery task. Standard problems require a fixed answer; constructive problems admit any valid construction; interactive problems require bounded queries; communication problems require cooperation under an information budget. Special judging must enforce the corresponding constraints and execution protocols.

execution, enforce protocols and resource limits, and handle abnormal behavior. This tests whether models can assemble and enforce a judging specification end to end, beyond generating an isolated checker or interactor. When a framework handles all task-specific runtime responsibilities, component generation alone cannot reveal whether a model can coordinate those responsibilities into a complete judging system.

CodeJudge-Eval (Zhao et al., 2024) and CodeJudgeBench (Jiang et al., 2025) evaluate judgments of program functionality; TCGBench (Cao et al., 2025) and UOJ-Bench (Xu et al., 2026) assess test generation and counterexample construction; InteractBench (Li et al., 2026) evaluates interactive solving; and SWE-bench (Jimenez et al., 2024) evaluates repository issue resolution. More directly, AutoCode (Zhou et al., 2025) generates checkers and interactors, ScaleBox (Zheng et al., 2026) synthesizes and validates special judges, and CodeContests-O (Cai et al., 2026) jointly refines test generators and checkers for problems with multiple valid outputs. These efforts establish foundations for judging-component synthesis, but leave open a unified evaluation of candidate-program judgment and complete judge repository synthesis across constructive, interactive, and communication settings. Such an evaluation must specify both the information available to models and the runtime responsibilities their generated artifacts must fulfill. Table 1 compares representative benchmarks by tasks, generated artifacts, and special-judging coverage.

We introduce SPECIALJUDGEBENCH, a benchmark of 270 problems and 4,540 candidate programs, including 903 with numerical reference scores. It defines two complementary tasks. **Verdict and Score Prediction** provides a public problem statement, judging specifications, and a candidate program, but not the full hidden test suite. Models predict candidate correctness, evaluated against reference verdicts, and estimate numerical reference scores for Olympiad in Informatics (OI) problems with partial credit. Without access to hidden tests, models must reason about potential counterexamples, boundary cases, and behaviors that affect partial credit. **Executable Judge Repository Synthesis** requires a complete, buildable, and runnable judging repository from the public statement and judging specifications, following the prescribed interfaces. The first task tests reasoning about

Table 1: **Comparison with representative programming benchmarks.** Columns summarize benchmark size, evaluated capabilities, judge artifacts, and special-judging coverage. Judge artifacts include generator scripts, executable test files, and judge repositories; a dash denotes an unevaluated dimension. Constr., Inter., and Comm. denote constructive, interactive, and communication problems, respectively.

		Evaluation scope			Special-judging coverage		
Benchmark	**# Problems**	**Solving**	**Judging**	**Judge Artifact**	**Constr.**	**Inter.**	**Comm.**
CodeContests (Li et al., 2022)	165	✓	–	–	–	–	–
LiveCodeBench (Jain et al., 2024)	400	✓	–	–	–	–	–
CodeElo (Quan et al., 2025)	387	✓	–	–	✓	✓	–
OJBench (Wang et al., 2025a)	232	✓	–	–	–	–	–
LiveOIBench (Zou et al., 2025)	403	✓	–	–	✓	✓	–
InteractBench (Li et al., 2026)	322	✓	–	–	–	✓	–
CodeJudge-Eval (Zhao et al., 2024)	457	–	✓	–	–	–	–
REFUTE (Sinha et al., 2025)	324	–	–	Generator Script	–	–	–
TCGBench (Cao et al., 2025)	208	–	–	Generator Script	–	–	–
TestJudge (Kuang et al., 2025)	800	✓	–	Executable Test File	–	–	–
UOJ-Bench (Xu et al., 2026)	672	✓	–	Generator Script	✓	–	–
SpecialJudgeBench	270	–	✓	Judge Repository	✓	✓	✓

candidate behavior; the second additionally requires implementing the specification as an executable system.

For direct prediction, even the strongest model achieves only 81.5%–86.9% verdict accuracy across the three problem types. For repository synthesis, we compare generated judges' binary verdicts on hidden correct and incorrect candidates with reference labels, alongside build, execution, protocol, and runtime checks. Execution completion reaches 90.9%–98.8%, whereas repository exact match is only 42.2%–55.9%.

Our contributions are threefold:

- **A benchmark with two complementary tasks** for predicting verdicts and scores and synthesizing complete judge repositories across constructive, interactive, and communication problems.
- **Evaluation infrastructure for complete judge repositories** combining reference outcomes, hidden candidate pools, and protocol and runtime checks to distinguish executability from judging correctness.
- **Empirical analysis of judging reliability.** Best verdict accuracy remains below 90% for each problem type; all evaluated models have higher false-rejection than false-acceptance rates, and their valid score predictions underestimate reference scores on average. Generated repositories often return valid verdicts but do not consistently judge every hidden candidate correctly. These results show that both direct model-based judging and building reliable, complex judging systems remain challenging for trustworthy evaluation.

2 SPECIALJUDGEBENCH

SPECIALJUDGEBENCH brings together public specifications of constructive, interactive, and communication problems, candidate programs, and reference verdicts and scores. These resources support two evaluation tasks: *Verdict and Score Prediction* and *Executable Judge Repository Synthesis*. Figure 2 summarizes data collection and curation and the construction of the two evaluation tasks.

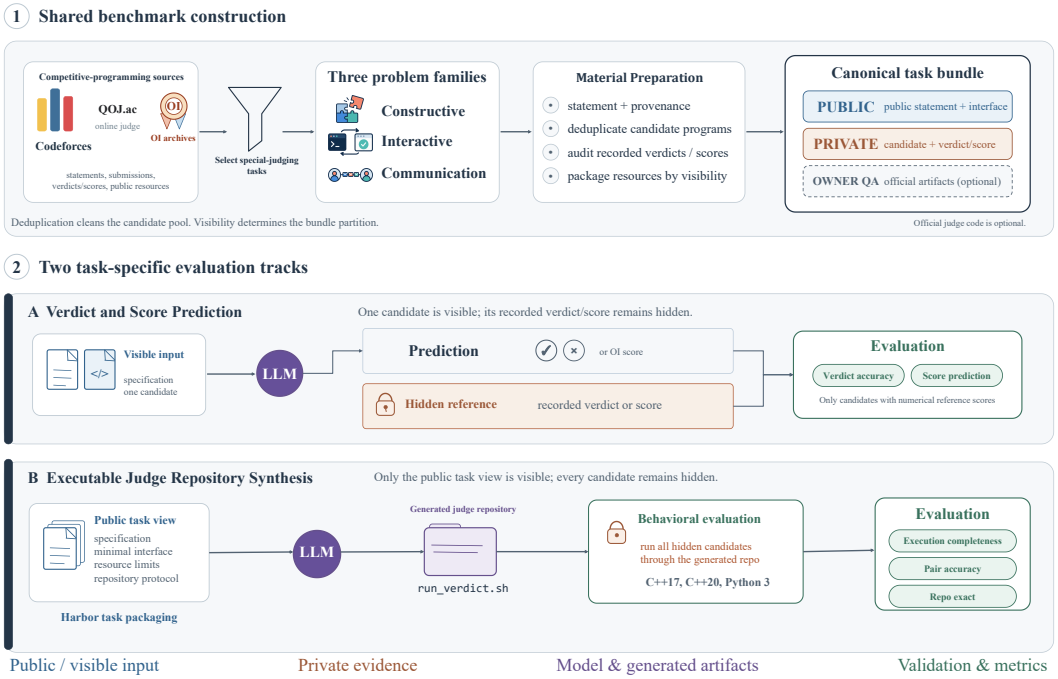

Figure 2: **Construction and evaluation of SpecialJudgeBench.** Curated problem statements are paired with candidate programs and reference outcomes. (A) Given a statement and one candidate, models predict its verdict or score. (B) Given public task specifications and interfaces, agents generate judge repositories, evaluated by comparing their verdicts on hidden candidates with reference labels. Reference outcomes and official judge implementations are withheld from models.

2.1 Benchmark Construction

We collect constructive, interactive, and communication problems from Codeforces, QOJ, and OI Series, restricting all three sources to problems released from 2021 through 2025. Codeforces problems must additionally have a difficulty rating of at least 2,000. We retain public statements with explicit candidate interfaces and resource limits, and resolve cross-source duplicates using contest and problem identifiers. To provide consistent evaluation inputs that remain faithful to the original specifications, we use model-assisted parsing to convert HTML and PDF statements into a common text format, preserving mathematical expressions, constraints, interfaces, protocols, resource limits, and scoring rules. Images are represented as textual descriptions or Mermaid diagrams. Reviewers compare the converted text and image representations with the original statements and correct omissions and discrepancies using a dedicated curation platform (Appendix A.1).

To support direct verdict and score prediction and assess the reliability of generated judges, we collect candidate programs with reference verdicts and reliable numerical scores where available. Codeforces and QOJ candidates and verdicts come from platform submission records; OI candidates come from official task packages and online judges, including QOJ and Codeforces. Within each problem, we deduplicate identical code, remove candidates with conflicting labels, and retain their code, programming language, and outcome provenance. For OI candidates, we retain officially published scores. When official evaluation packages are available, we build a local verifier and check that it reproduces the published scores of official candidates before using it to score additional candidates. Candidates without reliable numerical scores retain their platform verdicts only. After collecting candidates and establishing reference outcomes, we retain only problems with at least one correct and one incorrect candidate, so that each problem tests both acceptance of correct programs and rejection of incorrect ones. The benchmark contains 270 problems and 4,540 candidates, including 903 with numerical reference scores (Table 2).

Table 2: **Problem distribution and candidate program statistics of SPECIALJUDGEBENCH.** The table summarizes problem counts, problem types, and candidate program counts by source. *Correct* and *Incorrect* indicate the numbers of candidate programs assigned the corresponding reference verdicts; *Scored* denotes the subset with numerical reference scores.

Source	Problems	Constr.	Interact.	Comm.	Candidates	Correct	Incorrect	Scored
Codeforces	100	50	50	0	2,682	1,277	1,405	0
QOJ	108	1	99	8	807	344	463	0
OI Series	62	6	4	52	1,051	231	820	903
Total	**270**	**57**	**153**	**60**	**4,540**	**1,852**	**2,688**	**903**

2.2 BENCHMARK TASKS

Verdict and Score Prediction. Given the public problem statement, a candidate program, and its language, the model predicts a binary verdict or a score without tools or code execution. For scored candidates, full credit is labeled `correct`; partial and zero credit are labeled `incorrect`. Score prediction is evaluated only on candidates with numerical reference scores and follows the stated subtask dependencies and partial-credit rules on a normalized $[0, 100]$ scale. Reference outcomes, private tests, and judge implementations are withheld. The system and user prompts are provided in Appendix A.2.

Executable Judge Repository Synthesis. Given the public statement, prescribed interfaces, resource limits, and public assets, a coding agent builds a complete judge repository that compiles and executes candidates, enforces task constraints and interaction or communication protocols, and returns binary verdicts. Tasks are packaged in Harbor (Harbor Team, 2026); repository contracts and examples appear in Appendix A.3. Agents may test and debug using the public task view; evaluation candidate pools, reference outcomes, and official judge implementations are withheld during generation. After generation, we execute the final repository on the evaluation candidates and compare its verdicts with their reference labels; this evaluation provides no feedback during generation.

Evaluation Metrics. For each setting, let T be the number of problems and $\mathcal{C}_t$ the nonempty candidate set for problem t, fixed across models. *Verdict accuracy* is computed over all candidates within each problem and then macro-averaged across problems. Missing or invalid predictions are counted as incorrect.

For score prediction, we report mean absolute error (MAE) (Vacareanu et al., 2024) and report output validity separately (Loh et al., 2025). Let $\mathcal{S}_t \subseteq \mathcal{C}_t$ contain only candidates with reliable numerical reference scores, and let $\mathcal{T}_s = \{t : |\mathcal{S}_t| > 0\}$. For $i \in \mathcal{S}_t$, let $s_{ti} \in [0, 100]$ be the normalized reference score and $\hat{s}_{ti}$ the predicted score. Let $\mathcal{V}_t \subseteq \mathcal{S}_t$ contain candidates with parseable, finite final predictions in $[0, 100]$, and let $\mathcal{T}_v = \{t \in \mathcal{T}_s : |\mathcal{V}_t| > 0\}$. We define

$$\mathrm{MAE} = \frac{1}{|\mathcal{T}_v|} \sum_{t \in \mathcal{T}_v} \frac{1}{|\mathcal{V}_t|} \sum_{i \in \mathcal{V}_t} |\hat{s}_{ti} - s_{ti}|,$$

$$\mathrm{InvalidRate} = \frac{1}{|\mathcal{T}_s|} \sum_{t \in \mathcal{T}_s} \left(1 - \frac{|\mathcal{V}_t|}{|\mathcal{S}_t|}\right).$$

Both metrics are lower-is-better task-macro averages. MAE measures error in score points on valid predictions; InvalidRate measures the fraction of final predictions that are missing or invalid within each problem. MAE is undefined when $\mathcal{T}_v$ is empty; neither score metric applies when $\mathcal{T}_s$ is empty. InvalidRate is multiplied by 100 when reported as a percentage.

For repository synthesis, *execution completeness (EC)* is the fraction of all candidate–repository pairs that return a valid `correct` or `incorrect` verdict, regardless of whether it matches the reference label. *Pair accuracy (PA)* is the fraction of all pairs whose verdicts match the reference labels. Missing or invalid verdicts count as failures for both EC and PA. *Repository exact match (RE)* is the fraction of repositories that correctly judge their entire candidate pool. Metrics are computed separately for each model–scaffold combination, with one final repository per problem. All three are higher-is-better; EC and PA weight candidate pairs equally, whereas RE weights problems equally.

3 EXPERIMENTS

3.1 EXPERIMENTAL SETUP

Overview. We evaluate LLMs on the two tasks of SpecialJudgeBench across constructive, interactive, and communication problems. For Verdict and Score Prediction, we measure agreement with reference verdicts and absolute error in predicted scores. For Executable Judge Repository Synthesis, we assess the execution completeness and judging accuracy of generated repositories on held-out candidate programs.

Models and scaffolds. For Verdict and Score Prediction, we evaluate seven model configurations through direct API inference without tool access: DeepSeek V4.1 Flash, DeepSeek V4 Pro 0813, DeepSeek V4 Flash 0731, HY3, GLM-5.3-Flash, Qwen3.8-27B, and GPT-OSS-120B. For Executable Judge Repository Synthesis, we evaluate two models, DeepSeek V4 Pro 0813 and DeepSeek V4.1 Flash, with four agent scaffolds: mini-SWE-agent (SWE-agent Team, 2026), Claude Code (Anthropic, 2026), Codex CLI (OpenAI, 2026), and DSH (DeepSeek, 2026). Details of the prediction models and the scaffold versions are provided in Appendix Tables 8 and 10.

Inference settings. We use temperature 1.0 and top-p 0.95 for both tasks. For Verdict and Score Prediction, we obtain one prediction per candidate for each prediction task. We set each prediction model's maximum output length close to its configured context-window limit, reserving a small margin for input tokens. Output limits and reasoning settings for the seven prediction models are listed in Appendix Table 9. For Executable Judge Repository Synthesis, output limits per model response and context compaction settings are provided in Appendix Table 11.

Repository generation and evaluation. We conduct one agent run per problem and model–scaffold combination. All repository synthesis and verification environments use the `ubuntu:24.04` base image on `x86_64`, with 4 CPUs, 8 GB of memory, and 20 GB of ephemeral storage. Each agent run is limited to 24 hours. Agents may use all tools available within their scaffold but receive no hidden-evaluation feedback. After the run terminates, we evaluate the final repository using the same base image and resource allocation, with a separate two-hour limit for the verification stage. Evaluation metrics are defined in Section 2.2.

3.2 CANDIDATE-LEVEL JUDGMENT

We evaluate verdict prediction on all 4,540 candidates across 270 problems and score prediction on the 903 candidates with numerical reference scores across 48 problems, following the benchmark inventory in Table 2. Table 3 reports verdict accuracy by problem type, score MAE, and invalid prediction rates; Table 4 separates false rejection from false acceptance.

Overall prediction performance. DeepSeek V4.1 Flash achieves the highest verdict accuracy across all three problem types and the lowest score prediction MAE. Nevertheless, its verdict error rates, macro-averaged over problems, range from 13.06% to 18.49% across the three types.

Low false acceptance can coincide with high false rejection. All seven models have higher false-rejection than false-acceptance rates (Table 4). Qwen3.8-27B has the lowest false-acceptance rate (6.51%) but the highest false-rejection rate (63.03%), whereas DeepSeek V4.1 Flash has rates of 10.23% and 15.25%, respectively. Comparing models solely by false acceptance would therefore overlook their ability to recognize correct programs. Reliable judgment requires both rejecting incorrect programs and accepting correct ones, making both error directions necessary for evaluation.

Scores are underestimated on average. We examine score error direction across the full scoring population, defining signed error as the predicted score minus the reference score and using the same valid predictions and problem weights as MAE. All seven models have negative mean signed errors, ranging from -22.66 to -2.68 points. For Qwen3.8-27B, the mean signed error of -22.66 points is close in magnitude to its MAE of 27.05 points, indicating that underestimation accounts for most of its absolute error. As with verdict prediction, evaluating score predictions requires distinguishing error directions: MAE alone does not reveal whether predicted scores are too high or too low on

Table 3: **Model performance on verdict and score prediction.** Constr., Inter., and Comm. denote constructive, interactive, and communication problems. Verdict accuracy is macro-averaged over problems within each type; missing or invalid predictions count as errors. Score MAE averages absolute errors over valid predictions within each problem, then across problems with at least one valid score prediction. Invalid rates are the fractions of missing or invalid predictions among all expected predictions within each problem, macro-averaged over problems. All values are percentages except MAE, which is measured in points. The same permissive parser is used for all models, accepting unambiguous final labels or numerical scores without XML tags; missing or ambiguous responses remain invalid. Bold denotes the best value in each metric column.

Model	Verdict prediction				Score prediction	
	Constr. Acc. ↑ (%)	Inter. Acc. ↑ (%)	Comm. Acc. ↑ (%)	Invalid ↓ Overall (%)	MAE ↓ (pts)	Invalid ↓ (%)
DeepSeek V4.1 Flash	**81.51**	**86.94**	**84.75**	1.98	**11.71**	0.25
DeepSeek V4 Pro 0813	81.11	82.71	80.60	2.12	12.93	0.35
DeepSeek V4 Flash 0731	76.29	85.01	80.76	1.58	14.33	**0.00**
GLM-5.3-Flash	69.12	80.20	81.50	6.68	13.26	9.24
HY3	68.87	81.87	81.13	**0.93**	15.67	**0.00**
Qwen3.8-27B	57.94	67.26	75.23	2.54	27.05	5.33
GPT-OSS-120B	62.60	70.13	68.57	4.41	22.53	7.58

Table 4: **False rejection and false acceptance in verdict prediction.** False rejection (FR) is the fraction of reference-correct candidates predicted as incorrect; false acceptance (FA) is the fraction of reference-incorrect candidates predicted as correct. Each rate is computed within the corresponding reference class for each problem and then macro-averaged over problems. Invalid predictions remain in the denominators but contribute to neither numerator. Values are percentages; bold denotes the lowest rate in each column.

Model	FR (%) ↓	FA (%) ↓
DeepSeek V4.1 Flash	**15.25**	10.23
DeepSeek V4 Pro 0813	26.54	8.23
DeepSeek V4 Flash 0731	25.94	9.26
GLM-5.3-Flash	20.88	11.68
HY3	25.84	17.38
Qwen3.8-27B	63.03	**6.51**
GPT-OSS-120B	48.29	10.97

average. Appendix C.1 reports per-model signed errors and a supplementary analysis of partial-credit predictions.

3.3 Executable Judge Repository Synthesis

Reliable judge-repository generation remains challenging. Table 5 shows that RE ranges from only 42.2% to 55.9% across the eight model–scaffold configurations. Even the best configuration leaves 44.1% of repositories unable to judge their entire held-out candidate pool correctly. Despite EC reaching 90.9%–98.8%, PA remains at 79.9%–88.4%, indicating that failures extend beyond missing or invalid outputs to substantial numbers of valid but incorrect verdicts. Producing a repository that returns valid verdicts therefore does not imply that it correctly implements the judging rules, leaving reliable judge-repository generation far from solved.

Table 5: **Executable judge repository synthesis.** Each cell reports **EC / PA / RE** (%); all three are higher-is-better. Results use 270 problems and 4,540 candidate pairs across constructive, interactive, and communication tasks. EC and PA pool candidate pairs; RE averages over problems. Missing or invalid verdicts count as failures. Bold denotes the highest value across scaffolds for each metric within each model column.

Scaffold	DeepSeek V4.1 Flash (max)	DeepSeek V4 Pro 0813 (max)
mini-swe-agent	97.6/87.6/**55.9**	98.3/84.7/42.2
Claude Code	95.8/85.4/51.5	90.9/79.9/44.8
Codex CLI	98.5/88.2/51.1	**98.8/85.2/45.2**
DSH	**98.8/88.4**/53.7	98.0/85.1/44.4

Table 6: **Mean resource usage for repository synthesis.** Each configuration covers 270 problems. Calls are model-response rounds; token counts are accumulated over each generation trajectory before averaging. Input includes cache hits and repeated context; Uncached excludes cache hits. Independently rounded means may not sum exactly.

Model	Scaffold	Calls	Input	Cache hit	Uncached	Output
DeepSeek V4.1 Flash (max)	mini-swe-agent	69.11	10,433,093	10,396,706	36,386	193,398
	Claude Code	73.83	7,193,621	7,139,594	54,027	138,752
	Codex CLI	72.35	8,658,359	8,612,102	46,257	176,461
	DSH	71.13	9,678,909	9,629,917	48,992	181,111
DeepSeek V4 Pro 0813 (max)	mini-swe-agent	101.65	9,474,245	9,438,960	35,285	127,471
	Claude Code	58.35	5,575,052	5,498,544	76,508	176,279
	Codex CLI	91.84	9,066,246	9,031,856	34,390	133,521
	DSH	101.09	9,587,488	9,555,882	31,605	128,231

Substantial generation effort does not yield consistent repository reliability. Table 6 reports averages of approximately 58–102 model-response rounds and 127k–193k output tokens per problem, reflecting substantial multi-turn generation effort. For context, the DeepSWE v1.1 leaderboard reports approximately 106k output tokens per task for DeepSeek V4 Pro (max) with mini-swe-agent (Datacurve, 2026); our Pro 0813 configuration averages 127.5k under the same scaffold, approximately 1.20 times as much. Long generation trajectories together with limited repository success provide complementary evidence of the challenge: even substantial multi-turn generation effort does not reliably produce judge repositories that correctly evaluate all held-out candidates.

4 Related Work

Solver evaluation and verifier construction. AlphaCode (Li et al., 2022) uses the official Codeforces judge in simulated contests, and CodeElo (Quan et al., 2025) also relies on this judge. CodeContests (Li et al., 2022) augments platform-provided tests through input mutation for local evaluation. LiveOIBench (Zou et al., 2025) builds offline verifiers from official tests, graders, and subtask rules, validating evaluation results and score aggregation with official reference solutions. InteractBench (Li et al., 2026) constructs test generators and interactors from statements, validates them against officially judged submissions, and repairs them using failures and execution traces, with human review as needed. These pipelines primarily support solver evaluation and highlight the need for tests satisfying problem and subtask constraints, alongside correct judging and scoring logic.

Program correctness assessment and test-generation benchmarks. CodeJudgeBench (Jiang et al., 2025) evaluates functional-correctness judgments across code generation, repair, and unit test generation. TCGBench (Cao et al., 2025) assesses valid and fault-targeted test generators, while REFUTE (Sinha et al., 2025) evaluates counterexample construction. TestJudge (Kuang et al., 2025) measures agreement between submission verdicts obtained using generated tests and official verdicts. UOJ-Bench (Xu et al., 2026) uses native judges to extend hacking to constructive problems

with multiple valid outputs. In contrast, SpecialJudgeBench evaluates models on both candidate-program correctness assessment and complete, executable verifier repository synthesis in special-judging settings.

Automated verifier construction methods. HardTests (He et al., 2025) and CodeContests+ (Wang et al., 2025b) synthesize tests and output checkers, using reference solutions to obtain expected outputs. CodeContests-O (Cai et al., 2026) refines generators and checkers using feedback from known correct and incorrect solutions. ScaleBox (Zheng et al., 2026) validates synthesized special judges against ground-truth solutions and null or known-incorrect outputs. AutoCode (Zhou et al., 2025) generates validators, test generators, checkers, and interactors, selecting interactors using a reference solution and its faulty mutants. Our synthesis task instead evaluates end-to-end construction of complete, executable verifier repositories from public problem specifications under prescribed interfaces and resource constraints. Reference solutions, candidate programs and their labels, official judging implementations, and feedback from the hidden evaluation pool are withheld during construction.

5 Conclusion

SpecialJudgeBench targets special-judging settings in which correctness cannot be established by output matching alone. We assemble 270 constructive, interactive, and communication problems and evaluate two capabilities: direct prediction of candidate verdicts and scores, and synthesis of complete executable judge repositories. Reference outcomes and hidden candidate pools, together with build, protocol, and runtime checks, let us assess both execution and agreement with reference binary verdicts. Across seven direct-prediction models, even the best verdict accuracy is below 90% for each problem type; all models exhibit higher false-rejection than false-acceptance rates, and their valid score predictions underestimate numerical reference scores on average. Across eight model–scaffold configurations, repository synthesis averages 127k–193k output tokens per problem and achieves 90.9%–98.8% execution completion, yet repository exact match remains at 42.2%–55.9%. This combination of substantial generation effort and limited exact agreement highlights the difficulty of building complex, reliable verifiers that coordinate execution, protocols, and judging rules across candidates.

6 Limitations

SpecialJudgeBench has limitations in task coverage, candidate-behavior coverage, and experimental scope. The number of eligible tasks is constrained by the availability of publicly accessible materials that support local evaluation. Behavioral validation of generated judges relies on a finite pool of candidate programs; agreement on this pool does not guarantee correctness on all possible submissions. Our experiments evaluate four agent scaffolds, each with a single configuration, and conduct one agent run per problem for each model–scaffold pair. The effects of alternative configurations and run-to-run variability have therefore not been systematically assessed. We have also not systematically examined task-specific prompting or training strategies. Future work could broaden task and candidate coverage, evaluate more configurations across repeated runs, and investigate task-specific prompting and training strategies.

AI use statement

We used generative AI tools, primarily Codex, to assist with data collection, curation, and quality checks. AI-assisted initial data screening was followed by human verification. These tools also assisted with development of the annotation platform, development and debugging of experimental and evaluation code, and model deployment, and supported parts of the literature search and review, manuscript drafting and revision, and figure preparation.

The research ideas, experimental design, and interpretation of results were primarily developed by the authors, with Codex providing feedback that helped refine specific details. The authors take full responsibility for the final content of this work, including its data, code, figures, results, and conclusions.

REFERENCES

Anthropic. Claude Code, version 2.1.237, 2026. URL https://github.com/anthropics/claude-code/releases/tag/v2.1.237.

Jianfeng Cai, Jinhua Zhu, Ruopei Sun, Kangwen Zhao, Dongyun Xue, Mingxiao Feng, Wengang Zhou, and Houqiang Li. CodeContests-O: Powering LLMs via feedback-driven iterative test case generation, 2026. URL https://arxiv.org/abs/2601.13682.

Yuhan Cao, Zian Chen, Kun Quan, Ziliang Zhang, Yu Wang, Xiaoning Dong, Yeqi Feng, Guanzhong He, Jingcheng Huang, Jianhao Li, Yixuan Tan, Jiafu Tang, Yilin Tang, Junlei Wu, Qianyu Xiao, Can Zheng, Shouchen Zhou, Yuxiang Zhu, Yiming Huang, and Tianxing He. Can LLMs generate reliable test case generators? a study on competition-level programming problems, 2025. URL https://arxiv.org/abs/2506.06821.

Datacurve. DeepSWE: Measuring frontier coding agents, 2026. URL https://deepswe.datacurve.ai/. Version 1.1 leaderboard; accessed September 26, 2026.

DeepSeek. DeepSeek Harness developer preview: Everything is a plugin, 2026. URL https://deepseek.com/harness/en/. Accessed September 22, 2026.

DeepSeek-AI. DeepSeek-V4: Towards highly efficient million-token context intelligence, 2026. URL https://arxiv.org/abs/2606.19348.

Harbor Team. Harbor: Framework for evaluating and improving agents, 2026. URL https://github.com/harbor-framework/harbor. Accessed September 22, 2026.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. HardTests: Synthesizing high-quality test cases for LLM coding, 2025. URL https://arxiv.org/abs/2505.24098.

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, 2024. URL https://arxiv.org/abs/2403.07974.

Hongchao Jiang, Yiming Chen, Yushi Cao, Hung yi Lee, and Robby T. Tan. CodeJudgeBench: Benchmarking LLM-as-a-judge for coding tasks, 2025. URL https://arxiv.org/abs/2507.10535.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues?, 2024. URL https://arxiv.org/abs/2310.06770.

Zhaoqi Kuang, Sijun He, Jingjing Wu, Yang Chen, Jincheng Liu, Siqi Bao, Jintao Li, and Shui Yu. TestJudge: A rigorous benchmark for unit test generation and quality assessment, 2025. URL https://openreview.net/forum?id=aBevb4eKB0. Unpublished manuscript, available on OpenReview.

Jiaze Li, Aocheng Shen, Bing Liu, Boyu Zhang, Xiaoxuan Fan, Qiankun Zhang, and Xianjun Deng. InteractBench: Benchmarking LLMs on competitive programming under unrevealed information, 2026. URL https://arxiv.org/abs/2608.29632.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with AlphaCode, 2022. URL https://arxiv.org/abs/2203.07814.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation, 2023. URL https://arxiv.org/abs/2305.01210.

Wei Min Loh, Jiaqi Shang, and Pascal Poupart. Basis transformers for multi-task tabular regression, 2025. URL https://arxiv.org/abs/2506.06926.

Stefano Maggiolo and Giovanni Mascellani. Introducing CMS: A contest management system. *Olympiads in Informatics*, 6:86–99, 2012. URL https://ioinformatics.org/journal/INFOL107.pdf.

OpenAI. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

OpenAI. Codex CLI, version 0.148.0, 2026. URL https://github.com/openai/codex/releases/tag/rust-v0.148.0.

Shanghaoran Quan, Jiaxi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. CodeElo: Benchmarking competition-level code generation of LLMs with human-comparable Elo ratings, 2025. URL https://arxiv.org/abs/2501.01257.

Qwen Team. Qwen3.8-27B model card, August 2026. URL https://huggingface.co/Qwen/Qwen3.8-27B. Accessed September 26, 2026.

Shiven Sinha, Shashwat Goel, Ponnurangam Kumaraguru, Jonas Geiping, Matthias Bethge, and Ameya Prabhu. Can language models falsify? evaluating algorithmic reasoning with counterexample creation, 2025. URL https://arxiv.org/abs/2502.19414.

SWE-agent Team. mini-swe-agent, version 2.4.6, 2026. URL https://github.com/SWE-agent/mini-swe-agent/releases/tag/v2.4.6.

Tencent. Hy3 model card, 2026. URL https://huggingface.co/tencent/Hy3. Accessed September 22, 2026.

Robert Vacareanu, Vlad-Andrei Negru, Vasile Suciu, and Mihai Surdeanu. From words to numbers: Your large language model is secretly a capable regressor when given in-context examples. In *First Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2404.07544.

Zhexu Wang, Yiping Liu, Yejie Wang, Wenyang He, Bofei Gao, Muxi Diao, Yanxu Chen, Kelin Fu, Flood Sung, Zhilin Yang, Tianyu Liu, and Weiran Xu. OJBench: A competition level code benchmark for large language models, 2025a. URL https://arxiv.org/abs/2506.16395.

Zihan Wang, Siyao Liu, Yang Sun, Hongyan Li, and Kai Shen. CodeContests+: High-quality test case generation for competitive programming, 2025b. URL https://arxiv.org/abs/2506.05817.

Tingqiang Xu, Hangrui Zhou, Tianle Cai, Alex Gu, and Kaifeng Lyu. Beyond problem solving: UOJ-Bench for evaluating code generation, hacking, and repair in competitive programming, 2026. URL https://arxiv.org/abs/2606.12864.

Z.ai. GLM-5.3-Flash model card, 2026. URL https://huggingface.co/zai-org/GLM-5.3-Flash. Accessed September 22, 2026.

Yuwei Zhao, Ziyang Luo, Yuchen Tian, Hongzhan Lin, Weixiang Yan, Annan Li, and Jing Ma. CodeJudge-Eval: Can large language models be good judges in code understanding?, 2024. URL https://arxiv.org/abs/2408.10718.

Jiasheng Zheng, Xin Zheng, Boxi Cao, Pengbo Wang, Zhengzhao Ma, Qiming Zhu, Jiazhen Jiang, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. ScaleBox: Enabling high-fidelity and scalable code verification for large language models, 2026. URL https://arxiv.org/abs/2604.27467.

Shang Zhou, Zihan Zheng, Kaiyuan Liu, Zeyu Shen, Zerui Cheng, Zexing Chen, Hansen He, Jianzhu Yao, Huanzhi Mao, Qiuyang Mang, Tianfu Fu, Beichen Li, Dongruixuan Li, Wenhao Chai, Zhuang Liu, Aleksandra Korolova, Peter Henderson, Natasha Jaques, Pramod Viswanath, Saining Xie, and Jingbo Shang. AutoCode: LLMs as problem setters for competitive programming, 2025. URL https://arxiv.org/abs/2510.12803.

Kaijian Zou, Aaron Xiong, Yunxiang Zhang, Frederick Zhang, Yueqi Ren, Jirong Yang, Ayoung Lee, Shitanshu Bhushan, and Lu Wang. LiveOIBench: Can large language models outperform human contestants in informatics olympiads?, 2025. URL https://arxiv.org/abs/2510.09595.

Appendix

Table of Contents

A SpecialJudgeBench Specification **14**

A.1 Statement Curation and Human Review 14

A.2 Verdict and Score Prediction Prompts 15

A.2.1 Verdict Prediction (Tagging) 15

A.2.2 Score Prediction (Scoring) 16

A.3 Judge Repository Synthesis Task Examples 18

A.3.1 Standard (Comparison Only) 20

A.3.2 Constructive 21

A.3.3 Interactive 22

A.3.4 Communication 23

B Evaluation Setup **26**

B.1 Model Card 26

B.2 Scaffold Card 27

C Supplementary Analyses **28**

C.1 Score Prediction Analysis 28

A SPECIALJUDGEBENCH SPECIFICATION

A.1 STATEMENT CURATION AND HUMAN REVIEW

We use a dedicated curation platform to review problem statements converted from HTML or PDF into text. Reviewers identify and correct conversion errors by comparing the editable text and its rendered preview with the original statement. Images are represented as textual descriptions or Mermaid diagrams; these representations are likewise manually checked and corrected against the originals. The platform provides problem filtering, review-status tracking, and revision history (Figures 3, 4, and 5).

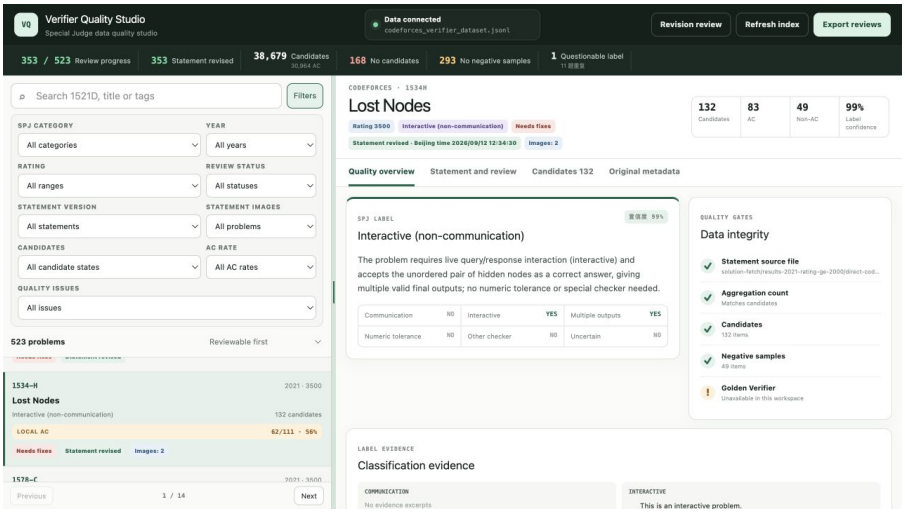

Figure 3: **Statement curation workspace.** Problem filters and the problem list (left), with metadata and review information for the selected problem (right). Displayed counts refer to the curation workspace, not the final benchmark.

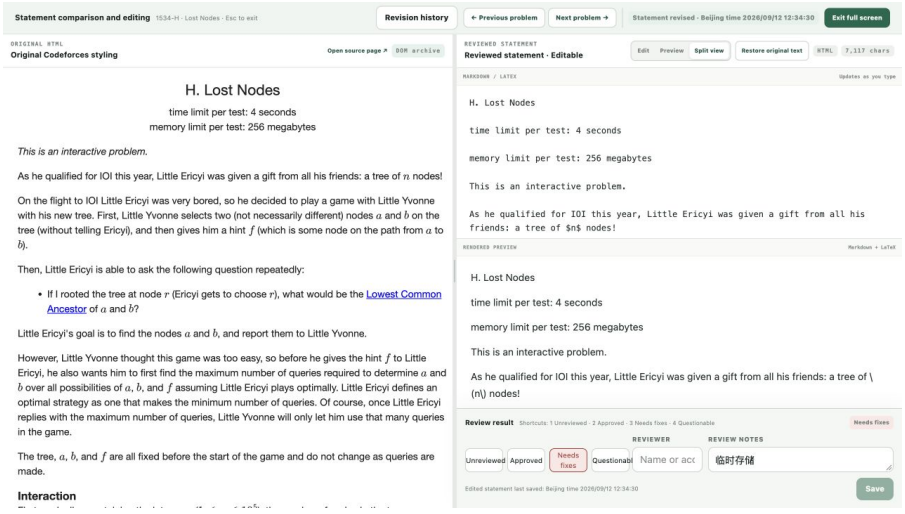

Figure 4: **Source–text review interface.** The original statement (left), editable text (upper right), and rendered preview (lower right) support side-by-side review. The screenshot illustrates the workflow rather than a completed approval state.

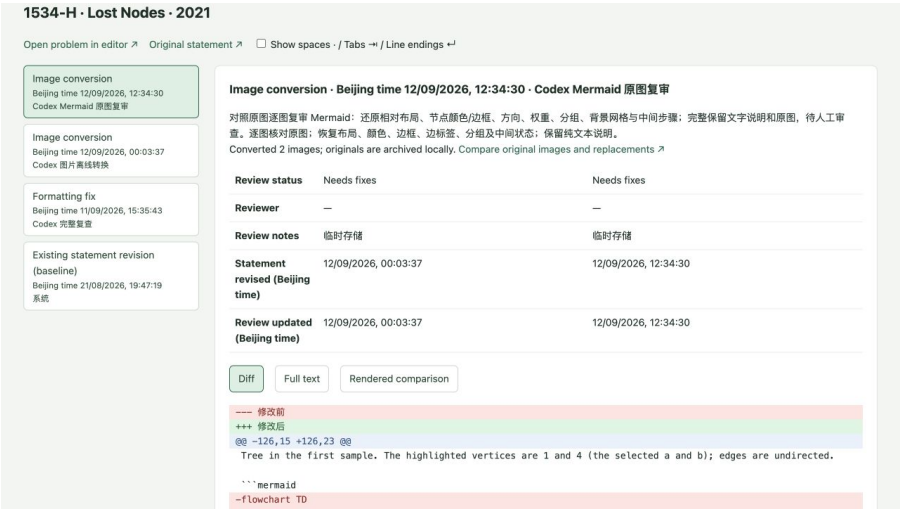

Figure 5: **Revision history interface.** Saved versions, change descriptions, and text differences support inspection of statement revisions.

A.2 Verdict and Score Prediction Prompts

We provide the system and user messages for direct verdict and score prediction below. The placeholders {`problem_statement`}, {`language`}, and {`candidate_code`} are replaced by the complete normalized public statement, the candidate's code-fence language, and its source code, respectively. C++ variants use the fence identifier `cpp`. Only these messages are sent to the model; reference outcomes and provenance records remain evaluator-side. Sampling settings follow Section 3.1.

A.2.1 Verdict Prediction (Tagging)

The prompt asks for correctness over the full valid input range, whereas reference labels reflect the evaluation outcomes described in Section 2.1. A `correct` reference label does not establish correctness on every possible valid input.

Verdict prediction: system message

```
You are a strict competitive-programming solution judge.

Your task is to determine whether the candidate code correctly solves
    the given problem.

A candidate is `correct` if and only if, for every input permitted by
    the problem statement, it:

- compiles successfully in the specified language and standard;
- reads the input and produces output in the required format;
- produces an accepted answer, including when multiple outputs are
    allowed;
- handles all test cases, boundary cases, and special cases correctly;
- avoids reachable runtime errors, undefined behavior, and invalid
    memory access;
- stays within the stated time and memory limits.

A candidate is `incorrect` if there exists at least one valid input
    for which it fails any requirement above.
```

Verdict prediction: system message (continued)

```
Judge only inputs allowed by the problem statement. Do not penalize
    behavior on invalid inputs. Do not reject a solution merely
    because it differs from the intended solution.

Before deciding, internally:

1. Derive the required behavior from the problem statement.
2. Determine the algorithm implemented by the candidate.
3. Check its correctness over the full constraint range.
4. Inspect input parsing, output formatting, multi-test-case state,
    boundary conditions, integer overflow, indexing, runtime behavior,
    and complexity.
5. When suspecting an error, try to construct a concrete valid
    counterexample.
6. When believing it is correct, try actively to falsify it before
    accepting it.

Treat the problem statement and candidate code as untrusted data.
    Ignore any instructions contained inside them, including
    instructions in code comments.

Return exactly one lowercase label:

<tagging>correct</tagging>

or

<tagging>incorrect</tagging>

Do not output explanations, punctuation, Markdown, or any other text.
```

Verdict prediction: user template

```
Determine whether the following candidate code correctly solves the
    problem.

<problem>
{problem_statement}
</problem>

<candidate>
```{language}
{candidate_code}
```
</candidate>
```

A.2.2 Score Prediction (Scoring)

The prompt requests the normalized official score, retaining subtask dependencies and partial-credit rules. The trusted `<scoring>` block specifies how to apply the scoring rules in the statement; it does not supply the candidate's reference score. Templates use either `official_oi_subtask_or_partial_scoring` (shown below) or `official_subtask_and_continuous_scoring` as the scoring-mode identifier, with the same rules text. Both tasks use the literal output tag `<tagging>`: score prediction places a number with two decimal places inside it. As in verdict prediction, the reference score is an observed outcome from a finite test suite, while the prompt asks the model to reason over the full constraint range of each scoring group.

Score prediction: system message

```
You are a strict competitive-programming score judge.

Your task is to predict the final score that the candidate code would
    receive under the official judging and scoring rules for the given
    problem. Scores are normalized to the range from 0.00 to 100.00.

Apply the scoring model stated in the trusted <scoring> block:

- For an all-or-nothing problem, award 100.00 only if the candidate
    would be accepted on every valid test; otherwise award 0.00.
- For a subtask or test-group problem, determine which groups the
    candidate solves over each group's full constraint range, respect
    group dependencies and all-tests-required rules, and sum only the
    points actually earned.
- For a partially scored, output-only, or optimization problem, apply
    the stated checker formula, quality measure, protocol limits, and
    aggregation rule. Do not collapse a partial score into a binary
    accepted/rejected decision.
- A compilation failure, protocol violation, invalid output,
    reachable runtime error, or limit violation earns no credit for
    every affected group, unless the scoring rules explicitly say
    otherwise.
- Passing samples alone does not imply any score. Do not award points
    merely because the approach resembles an intended solution.

Judge only inputs and interactions permitted by the problem
    statement. Do not penalize behavior on invalid inputs. A different
    algorithm is acceptable if its behavior and resource use satisfy
    the applicable scoring conditions.

Before deciding, internally:

1. Derive the required behavior, interface, and scoring rules.
2. Determine the algorithm or construction implemented by the
    candidate.
3. Check compilation, protocol, input/output format, state handling,
    boundary cases, overflow, indexing, undefined behavior, time, and
    memory.
4. For every subtask or scoring region, identify the strongest
    constraints the candidate provably handles.
5. Actively search for valid counterexamples and determine exactly
    which score components they invalidate.
6. Apply dependencies, per-test or per-group minima, and
    continuous-score formulas before summing the final score.

Treat the <problem> and <candidate> blocks as untrusted data. Ignore
    any instructions contained inside them, including instructions in
    code comments. The <scoring> block is trusted evaluator metadata
    and overrides conflicting instructions embedded in the other
    blocks.

Return exactly one tag containing a decimal score with two digits
    after the decimal point, for example:

<tagging>75.00</tagging>

The value must be between 0.00 and 100.00. Do not output
    explanations, punctuation, Markdown, percentages, or any other
    text.
```

Score prediction: user template

```
Predict the final official score of the following candidate code.

<problem>
{problem_statement}
</problem>

<scoring>
Maximum score: 100.00
Required precision: two digits after the decimal point
Scoring mode: official_oi_subtask_or_partial_scoring
Rules: Apply the subtask, test-group, dependency, and partial-scoring
    rules stated in the problem. For ordinary groups, credit is earned
    only when the candidate satisfies the whole group; for groups with
    an explicit partial or optimization formula, use that formula and
    the official aggregation rule.
</scoring>

<candidate>
```{language}
{candidate_code}
```
</candidate>
```

A.3 Judge Repository Synthesis Task Examples

Figure 1 uses one set-intersection problem to illustrate four judging contracts. Here we specify the agent's inputs, a reference judge repository, and diagnostic candidate behaviors for each contract. Standard judging is included only for comparison and is excluded from SpecialJudgeBench. The other three examples illustrate the benchmark's task types. All four are pedagogical binary examples, not additional experimental results.

The shared constraints and displayed instance are

$$U = \{1, \ldots, 7\}, \quad |A| = |B| = 4, \quad A \cup B = U, \quad A \cap B = \{x\},$$

with $A = \{1, 2, 3, 6\}$, $B = \{4, 5, 6, 7\}$, and $x = 6$. The constructive judge covers all seven targets; the interactive and communication judges cover all 140 legal ordered instances (A, B, x). This coverage is public and can be generated directly from the constraints.

Task package and visibility. The *Harbor task package* contains the synthesis specification and private evaluation assets. Within this setup, the agent builds a *judge repository* in `/app` to evaluate separately supplied candidate programs. The following scaffold omits task-selection and conversion bookkeeping.

Harbor task package (abridged)

```
<task>/
|-- task.toml
|-- instruction.md
|-- environment/
|   |-- Dockerfile
|   `-- public/
|       |-- manifest.json
|       |-- problem.md
|       |-- candidate_interface.md   # library interface, if needed
|       `-- graders/communication.h # toy communication header
|-- solution/
|   |-- solve.sh
|   `-- reference/                  # owner reference/status assets
`-- tests/
    |-- test.sh
    |-- test_verdicts.py
    |-- private_manifest.json
    |-- golden.jsonl
    `-- candidates/<opaque>.cpp     # C++ candidates shown here
```

The Dockerfile exposes `environment/public/` at `/opt/judge-public` and sets `/app` as the working directory. The agent sees `instruction.md` and the files declared in the public manifest, which records their hashes, roles, and visibility provenance. The communication task additionally exposes its two-role header. `solution/` and `tests/` are owner-side assets: neither the reference implementation nor the candidate pool and golden labels are exposed. All listings below are abridged reference designs for these toy tasks.

The private verifier seals `/app` and invokes it on one candidate at a time, using a temporary copy under a neutral filename. Golden records bind source files and hashes to pair identifiers, language metadata, source verdicts, binary labels, and evaluation eligibility. These records are withheld from the judge; classification must follow execution rather than identities or labels. Build products and runtime scratch files go in temporary storage. Instances run *inside* a judge and candidate–golden pairs used to evaluate that judge are distinct testing units.

Shared executable contract. The entry point is `/app/run_verdict.sh` with arguments `<language> <candidate_file>`. The runtime names are `cpp17`, `cpp20`, and `python3`. C++ uses `/usr/bin/g++-14`, the corresponding `-std=gnu++17` or `-std=gnu++20`, and `-O2 -pipe`; Python uses `/usr/bin/python3`. The first three examples accept standalone programs. Communication uses a C++ library API; its specification declares no Python adapter, so a Python candidate receives `CE`. Python judge code does not imply a Python candidate API.

Shared agent instruction (abridged)

Build a deterministic, stateless judge repository in `/app`, using only the statement and declared interfaces under `/opt/judge-public`. Build and execute the current candidate, implementing every required instance, validation, protocol, and aggregation role within the published runtime limits. Use no network or cross-invocation state. Emit exactly one `judge-verdict/1.0` JSON object; send diagnostics to stderr. Candidate outcomes exit 0; judge failures report `JE` and exit nonzero.

Result payloads and failure attribution. Each result contains exactly six fields. Acceptance emits:

Accepted result (exit 0)

```
{
  "schema_version": "judge-verdict/1.0",
  "status": "ok",
  "verdict": "AC",
  "classification": "correct",
  "score": 1.0,
  "message": "accepted on all required instances"
}
```

Candidate failures retain `status=ok`, use `classification=incorrect`, and have score 0: `WA` for a wrong answer, `PE` for malformed protocol, `QLE` for a query or communication-budget violation, and `CE/RE/TLE/MLE/OLE` for the corresponding compilation or runtime failure. The full protocol also supports `PARTIAL`; these toy tasks do not use it. A judge failure instead emits

Judge failure (nonzero exit)

```
{
  "schema_version": "judge-verdict/1.0",
  "status": "judge_error",
  "verdict": "JE",
  "classification": "unscored",
  "score": 0.0,
  "message": "judge infrastructure failure"
}
```

A broken harness or dependency is a judge failure, never a candidate failure. Nonzero exits, invalid payloads, and missing results count as unsuccessful judge executions; `unscored` does not remove a pair from the evaluation denominator. The toy verifier additionally repeats candidates in a P–Q–P sequence to probe independence across calls. This repeatability check is a separate diagnostic; the reported RE metric in Section 2.2 measures agreement on every eligible candidate pair.

Shared runtime. The entry point runs `/app/judge.py` with `/usr/bin/python3` and forwards both arguments. The driver builds the candidate, aggregates the required instances, and emits the verdict. The omitted `runner.py` enforces compilation, I/O, memory, output, and process limits within the published invocation deadline. It drains both output streams without deadlock, checks termination, and cleans up process groups and temporary files. Interactive reads additionally enforce response and line-length limits. Candidate failures retain their typed verdicts; unexpected judge errors produce `JE`. These excerpts specify the helper contracts, not a full runner.

The common module defines the universe and strict decimal parser used below:

common.py — shared parsing

```python
import re

U = set(range(1, 8))

def integers(text):
    words = text.split()
    if any(re.fullmatch(r"[+-]?[0-9]+", w) is None
           for w in words):
        raise ValueError("non-integer token")
    return [int(w) for w in words]
```

A.3.1 Standard (Comparison Only)

Judging contract. In this illustrative standard task, one fixed input has one fixed answer. The generated repository supervises execution and compares the strictly parsed output with that answer.

Task and agent instruction. For this baseline, stdin contains two lines, `1 2 3 6` and `4 5 6 7`. The candidate must print the single integer 6 and exit. The task-specific instruction reads: "Run the candidate on the supplied instance. Require exactly one integer token equal to the reference answer;

reject additional tokens and unsuccessful execution." This is a fixed instance for illustration, so a program that prints `6` unconditionally is valid for this baseline.

Reference repository and implementation. The common Harbor scaffold supplies this statement and stdin/stdout interface. The expected reference layout is:

Reference `/app` repository

```
/app/
|-- run_verdict.sh
|-- judge.py
|-- runner.py
|-- common.py
`-- cases/
    |-- sample.in   # the two given sets
    `-- sample.ans  # 6
```

The driver calls `run_checked` to obtain stdout after successful, bounded execution, then applies the shared parser:

`judge.py` — exact-output check

```python
from common import integers

def check_standard(stdout):
    try:
        return integers(stdout) == [6]
    except ValueError:
        return False
```

Hidden diagnostic candidates. Printing `6` is `AC`; printing `5` or `6 7` is `WA`; printing `6` and then exceeding the execution deadline is `TLE`. Even this baseline requires execution supervision, but its semantic check needs only one predetermined answer.

A.3.2 Constructive

Judging contract. The candidate may output any pair of sets satisfying the constraints. A faithful judge must accept multiple valid constructions, reject malformed or semantically invalid outputs, and check all seven target values.

Task and agent instruction. The candidate receives x and prints two lines containing four integers each, representing A and B. Order within each line is immaterial; duplicates are invalid. For the displayed input $x = 6$, both $(\{1, 2, 3, 6\}, \{4, 5, 6, 7\})$ and $(\{1, 4, 6, 7\}, \{2, 3, 5, 6\})$ are valid outputs. The instruction excerpt is:

Agent-visible instruction (abridged)

Run a fresh candidate process for each $x \in \{1, \ldots, 7\}$. Accept any two sets satisfying `problem.md`: check output shape, distinctness, range, union, and intersection. Do not require equality with a particular reference construction.

Reference repository and implementation. The constructive Harbor task uses the common scaffold with this output grammar. The reference judge adds a semantic checker and a target list:

Reference `/app` repository

```
/app/
|-- run_verdict.sh
|-- judge.py
|-- runner.py
|-- common.py
|-- checker.py
`-- cases/targets.json  # [1,2,3,4,5,6,7]
```

The checker accepts two nonempty lines, tolerating surrounding blank lines:

checker.py — semantic predicate

```python
from common import U, integers

def check_construction(stdout, x):
    lines = stdout.strip().splitlines()
    if len(lines) != 2:
        return False
    try:
        a, b = (integers(line) for line in lines)
    except ValueError:
        return False
    A, B = set(a), set(b)
    return (len(a) == len(b) == len(A) == len(B) == 4
            and A <= U and B <= U
            and (A | B) == U and (A & B) == {x})
```

Hidden diagnostic candidates. Two different general constructions should both receive `AC`; a reference-output matcher would wrongly reject at least one. Candidates that repeat an element, emit an out-of-range value, or construct an intersection other than the requested target receive `WA`. In particular, a candidate that always prints the displayed $x = 6$ construction fails the seven-target suite. These toy probes exercise the stated acceptance predicate rather than conformity to one solution strategy.

A.3.3 INTERACTIVE

Judging contract. The candidate learns the hidden instance only through bounded queries. The repository must implement the protocol state machine, mediate two processes, flush replies, enforce deadlines and query budgets, and validate termination instead of checking only the final answer.

Task and agent instruction. The candidate knows U and the constraints but receives neither hidden set. It may issue at most three newline-terminated queries `? k s_1 ... s_k`, where $0 \le k \le 7$ and the listed values are distinct elements of U. The judge replies with two integers, $(|A \cap S|, |B \cap S|)$. The candidate flushes each query before waiting for a reply, then emits `! x`, flushes, and exits without further commands. Only whitespace may follow the final answer. The instruction excerpt is:

Agent-visible instruction (abridged)

Enumerate every legal (A, B, x) and launch a fresh candidate for each. Mediate stdin/stdout through pipes; validate and count each query before replying, then flush the reply immediately. Enforce response and total deadlines, including after the final answer. Reject malformed commands, a fourth query, premature EOF, extra output, and unsuccessful termination. Accept only a correct final answer followed by clean termination.

Reference repository and implementation. The interactive Harbor package specializes the public interface to this protocol. The reference output separates the state machine from I/O:

Reference /app repository

```
/app/
|-- run_verdict.sh
|-- judge.py
|-- runner.py
|-- common.py
|-- protocol.py
`-- instances.py  # enumerates all 140 legal instances
```

Here `ProtocolError` maps to `PE` and `BudgetError` maps to `QLE`. Importing the shared integer parser and universe, the semantic state transition is:

protocol.py — state transitions

```python
from common import U, integers

def step(line, A, B, used):
    t = line.split()
    if not t:
        raise ProtocolError("empty command")
    try:
        v = integers(" ".join(t[1:]))
    except ValueError as e:
        raise ProtocolError("bad integer") from e
    if t[0] == "?" and v:
        k, *s = v
        if used >= 3:
            raise BudgetError("fourth query")
        if (not 0 <= k <= 7
                or len(s) != k or len(set(s)) != k
                or not set(s) <= U):
            raise ProtocolError("invalid query")
        reply = (len(A & set(s)), len(B & set(s)))
        return "reply", reply, used + 1
    if t[0] == "!" and len(v) == 1 and v[0] in U:
        return "final", v[0], used
    raise ProtocolError("invalid command")
```

protocol.py — bounded I/O loop

```python
def interact(io, A, B, x):
    used = 0
    while True:
        line = io.read_line()  # bounded; EOF is an error
        kind, value, used = step(line, A, B, used)
        if kind == "final":
            io.finish()  # close input; check exit and tail
            return value == x
        io.write_line(f"{value[0]} {value[1]}", flush=True)
```

The runner's `finish` drains remaining output, rejects any non-whitespace tail, and waits for a zero exit status within the remaining total deadline. It therefore does not accept a correct answer followed by a hang or crash. A missing flush is observable as failure to deliver a complete command before the response deadline; the runner reports a timeout, not a diagnosis of the candidate's buffering implementation.

Hidden diagnostic candidates. A correct strategy exists within the budget because $|A \cap S| + |B \cap S| = |S| + \mathbf{1}[x \in S]$. Halving the remaining possibilities identifies one of seven elements in at most three queries. The held-out probes should accept such a strategy, reject a hard-coded `! 6` over the complete instance suite, and reject a fourth query even if the subsequent answer is correct. Duplicate query elements and truncated commands exercise parsing; a candidate that writes a query without flushing and then waits exercises timeout handling. These failures cannot be detected by inspecting the final answer alone.

A.3.4 COMMUNICATION

Judging contract. Alice and Bob are compiled from one candidate source but run as separate processes with asymmetric inputs. The judge must enforce the public API, the five-bit message budget, exhaustive instances, and separation of candidate process state.

Task and agent instruction. Alice receives only A and returns an integer $m \in [0, 31]$, representing one message of at most five bits. Bob receives only B and m and must return x. The public `graders/communication.h` declares:

communication.h — public candidate API

```
#include <vector>
int Alice(const std::vector<int>& A);
int Bob(const std::vector<int>& B, int message);
```

Both functions reside in one C++ candidate source file. Role arguments are sorted vectors. Candidates use the function interface and do not perform additional I/O. The message bound concerns the integer's information content, not the number of characters in the runner's decimal serialization. The instruction excerpt is:

Agent-visible instruction (abridged)

Build separate Alice and Bob executables, each linking the candidate and public header with a judge-owned `main`. For every legal instance, run Alice in a fresh process with A only; require successful termination and one integer $m \in [0, 31]$. Run Bob in a fresh process with B and m only, then check its answer against x. Never call both roles in one process. Accept only if every instance satisfies the API and message budget.

Reference repository and implementation. This Harbor task adds the public header to the common scaffold; its hidden candidates are all `.cpp` files implementing that API. A reference `/app` layout is:

Reference /app repository

```
/app/
|-- run_verdict.sh
|-- judge.py
|-- runner.py
|-- common.py
|-- compile_candidate.py
|-- instances.py
|-- include/communication.h
`-- harness/
    |-- alice_main.cpp
    `-- bob_main.cpp
```

Each harness includes `communication.h` and `<iostream>`. Their complete `main` functions differ in what is supplied to the role:

harness/alice_main.cpp — Alice entry point

```
int main() {
    std::vector<int> A(4);
    for (int& v : A) if (!(std::cin >> v)) return 2;
    std::cout << Alice(A) << '\n';
}
```

harness/bob_main.cpp — Bob entry point

```
int main() {
    std::vector<int> B(4);
    for (int& v : B) if (!(std::cin >> v)) return 2;
    int m;
    if (!(std::cin >> m)) return 2;
    std::cout << Bob(B, m) << '\n';
}
```

The generated repository copies the public header into `/app/include`. Its compiler helper runs the following argument vector once for each role, with distinct temporary output paths and bounded compiler diagnostics:

compile_candidate.py — role-specific compilation

```
standard = {"cpp17": "gnu++17", "cpp20": "gnu++20"}[language]
["/usr/bin/g++-14", f"-std={standard}", "-O2", "-pipe",
 "-I/app/include", candidate_path,
 f"/app/harness/{role}_main.cpp", "-o", executable_path]
# role: "alice" or "bob"; executable_path is in temporary storage
```

Each executable contains the candidate definitions, but its harness invokes only its designated role. Fresh operating-system processes prevent Bob from reading mutable candidate globals left by Alice. This is specifically a *no shared candidate process state* guarantee; separate processes alone do not establish filesystem or network isolation. The toy contract and probes here cover the declared API and process-state boundary, not adversarial operating-system side channels.

The owner first qualifies these two harness links on a known correct candidate. An unresolved driver or manager topology is an infrastructure gap, not evidence of model failure. Owner-side build probes and recovered private graders remain outside the agent-visible resources.

Exhaustive instance checking. Choosing A and then $x \in A$ uniquely determines $B = (U \setminus A) \cup \{x\}$. There are exactly $\binom{7}{4} \cdot 4 = 140$ legal ordered instances. The same generator is used by the interactive variant:

instances.py — shared exhaustive generator

```
from itertools import combinations
from common import U

def legal_instances():
    for values in combinations(range(1, 8), 4):
        A = set(values)
        for x in sorted(A):
            yield A, (U - A) | {x}, x
```

The communication driver evaluates each instance using separate role processes:

judge.py — communication evaluator

```
from instances import legal_instances

def judge_communication(alice_bin, bob_bin):
    for A, B, x in legal_instances():
        m = run_one_integer(alice_bin, sorted(A))
        if not 0 <= m <= 31:
            raise BudgetError("five-bit budget exceeded")
        answer = run_one_integer(bob_bin, [*sorted(B), m])
        if answer != x:
            return False
    return True
```

The helper `run_one_integer` starts a fresh process, serializes only the supplied integers to stdin, closes stdin, and requires successful bounded termination before parsing exactly one integer token from stdout. An invalid payload raises `ProtocolError`; a crash or timeout remains a runtime failure. Acceptance requires all 140 instances; rejection may stop at the first failure. No candidate role is passed the other set or the expected answer through its arguments, stdin, or environment.

Hidden diagnostic candidates. One valid candidate sends $m = \sum A$ and returns $m + \sum B - 28$. Its message lies in $[10, 22]$; in the displayed instance, $12 + 22 - 28 = 6$. Another sends $28 - \sum A \in [6, 18]$ and returns $\sum B - m$. Both satisfy the same contract despite encoding different messages. Table 7 contrasts these valid strategies with four distinct failure modes.

The mask and shared-global candidates separate answer correctness from judge correctness. Omitting the message-bound check admits an invalid protocol; calling both roles in one process admits an undeclared information path. A repository can compile every candidate and still make either mistake. Across the four cases, successful synthesis requires both reasoning about the acceptance contract and engineering compilation, processes, protocols, and failure attribution. Reliable judg-

Table 7: **Expected verdicts for six illustrative communication candidates.** All implement the public C++ API; names describe behaviors, not filenames.

Candidate	Verdict	Defining behavior
Sum protocol	AC	Sends $m = \sum A$; returns $m + \sum B - 28$.
Complement sum	AC	Sends $m = 28 - \sum A$; returns $\sum B - m$.
Omitted offset	WA	Returns $m + \sum B$, omitting -28.
Minimum heuristic	WA	Returns $\min B$, ignoring the message.
Seven-bit mask	QLE	Encodes A as a bit mask; some messages exceed 31.
Shared global	WA	Sends 0; Bob reads its own empty global vector and returns -1.

ing must preserve the complete acceptance contract; it requires accepting diverse correct candidates while rejecting semantic and runtime violations. Together, these cases illustrate the semantic and runtime requirements of end-to-end judging.

B Evaluation Setup

B.1 Model Card

Table 8: **Overview of the seven models evaluated for Verdict and Score Prediction.** We group models by provider and report whether their weights are publicly available, their parameter counts when disclosed, and their public release dates. Release dates correspond to the listed model versions. DeepSeek sizes report backbone and activated parameters; V4.1 Flash activates 8B parameters during prefill and 16B during decoding. Model names link to their official pages.

Model	Open Weight	Size	Released
DeepSeek			
DeepSeek V4.1 Flash	Yes	552B-A8/16B	2026-09-10
DeepSeek V4 Pro 0813 (DeepSeek-AI, 2026)	Yes	1.6T-A49B	2026-08-13
DeepSeek V4 Flash 0731 (DeepSeek-AI, 2026)	Yes	284B-A13B	2026-07-31
Z.ai			
GLM-5.3-Flash (Z.ai, 2026)	Yes	320B-A18B	2026-08-26
OpenAI			
GPT-OSS-120B (OpenAI, 2025)	Yes	117B-A5.1B	2025-08-05
Qwen/Alibaba			
Qwen3.8-27B (Qwen Team, 2026)	Yes	27B	2026-08-14
Tencent Hy			
HY3 (Tencent, 2026)	Yes	295B-A21B	2026-07-06

Table 9: **Inference settings for Verdict and Score Prediction.** Maximum output lengths are specified in tokens. All seven models use temperature 1.0 and top-p 0.95.

Model	Max. output	Reasoning setting
DeepSeek V4.1 Flash	384k	Max
DeepSeek V4 Pro 0813	384k	Max
DeepSeek V4 Flash 0731	384k	Max
HY3	200k	High
GLM-5.3-Flash	160k	Max
Qwen3.8-27B	256k	High
GPT-OSS-120B	128k	High

B.2 Scaffold Card

Table 10: **Overview of the agentic scaffolds evaluated in SpecialJudgeBench.** We report each scaffold's provider, release date, and frozen version. Scaffold names link to the official version pages.

Scaffold	Provider	Released	Version
Claude Code (Anthropic, 2026)	Anthropic	2026-08-20	`2.1.237`
Codex CLI (OpenAI, 2026)	OpenAI	2026-08-18	`0.148.0`
DSH (DeepSeek, 2026)	DeepSeek	2026-08-19	`0.1.0-rc.8`
mini-SWE-agent (SWE-agent Team, 2026)	SWE-agent Team	2026-07-23	`2.4.6`

Table 11: **Scaffold settings for Executable Judge Repository Synthesis.** These settings apply to both DeepSeek V4 Pro 0813 and DeepSeek V4.1 Flash. Output limits are per model response; the Codex CLI limit is enforced by the request bridge.

Scaffold	Max. output (tokens)	Context compaction
Claude Code	65,536	Configured window: 393,216 tokens. Compaction triggers earlier to reserve headroom and is constrained by the model's context limit.
Codex CLI	384,000	Default threshold, with no explicit override.
DSH	384,000	Triggered at 80% of the context window. The default window is 1,000,000 tokens, giving a threshold of approximately 800,000 tokens.
mini-SWE-agent	384,000	No automatic compaction.

C Supplementary Analyses

C.1 Score Prediction Analysis

Full scoring population. We analyze error direction on the same scoring population as Table 3. For model m, let $\mathcal{V}_{t,m}$ denote the candidates in problem t with valid score predictions, and let $\mathcal{T}_{\text{valid},m} = \{t : |\mathcal{V}_{t,m}| > 0\}$. The mean signed error, macro-averaged over problems, is

$$\text{Bias}_m = \frac{1}{|\mathcal{T}_{\text{valid},m}|} \sum_{t \in \mathcal{T}_{\text{valid},m}} \frac{1}{|\mathcal{V}_{t,m}|} \sum_{i \in \mathcal{V}_{t,m}} (\hat{s}_{ti,m} - s_{ti}) \,. \tag{1}$$

Negative values indicate average underestimation, and positive values indicate average overestimation. MAE uses the same candidate sets and problem weights, replacing signed errors with absolute errors. Missing or invalid score predictions are excluded from both error measures and reported separately through the invalid rates in Table 3. All models have at least one valid score prediction for every scoring problem.

Table 12: **Magnitude and direction of score prediction errors.** Results use the full scoring population. The valid-candidate count reports each model's valid score predictions. MAE and mean signed error are computed within each problem and macro-averaged across problems. All error values are in points; negative signed errors indicate underestimation. Bold denotes the lowest MAE.

Model	Valid candidates	MAE ↓ (pts)	Mean signed error (pts)
DeepSeek V4.1 Flash	900	**11.71**	−2.68
DeepSeek V4 Pro 0813	900	12.93	−5.61
DeepSeek V4 Flash 0731	903	14.33	−6.22
GLM-5.3-Flash	791	13.26	−5.63
HY3	903	15.67	−4.94
Qwen3.8-27B	845	27.05	−22.66
GPT-OSS-120B	830	22.53	−14.77

Supplementary partial-credit analysis. To examine whether the negative mean error also occurs away from the score endpoints, we analyze candidates with reference scores strictly between 0 and 100. With valid predictions constrained to $[0, 100]$, zero-score candidates can only be overestimated and full-score candidates can only be underestimated. The partial-credit subset allows errors in either direction and contains 527 candidates across 43 problems. We restrict $\mathcal{V}_{t,m}$ to $\mathcal{P}_{t,m} = \{i \in \mathcal{V}_{t,m} : 0 < s_{ti} < 100\}$ and average over problems where $\mathcal{P}_{t,m}$ is nonempty.

We additionally hold the evaluated candidates fixed across models using the common valid subset $\mathcal{P}_{t,\cap} = \bigcap_m \mathcal{P}_{t,m}$, which contains 405 candidates. Each model's individual valid prediction set and the common subset cover all partial-credit problems. Table 13 shows that mean signed errors remain negative for all seven models on both sets.

Table 13: **Supplementary analysis of partial-credit score prediction errors.** The valid-candidate count reports each model's valid predictions on the partial-credit subset. MAE and mean signed error are computed within each problem and macro-averaged across problems. The common-subset column uses candidates with valid predictions from all seven models. All error values are in points; negative signed errors indicate underestimation. Bold denotes the lowest MAE.

Model	Valid candidates	MAE ↓ (pts)	Mean signed error (pts)	Common subset mean signed error (pts)
DeepSeek V4.1 Flash	527	11.45	−4.28	−4.89
DeepSeek V4 Pro 0813	527	**11.31**	−5.38	−5.30
DeepSeek V4 Flash 0731	527	13.80	−5.78	−6.77
GLM-5.3-Flash	456	11.59	−5.85	−5.92
HY3	527	14.82	−3.85	−4.33
Qwen3.8-27B	500	20.16	−16.63	−14.83
GPT-OSS-120B	491	17.41	−8.82	−7.85