RSI-FORGE: FROM RESEARCH PAPERS TO ENVIRONMENTS FOR RECURSIVE SELF-IMPROVEMENT

Anonymous authors
Paper under double-blind review

ABSTRACT

Environments are the foundation of recursive self-improvement: they provide the problems agents work on and the feedback used to evaluate progress. Yet constructing challenging research environments with reliable evaluation still depends on domain experts, limiting their scale and disciplinary coverage. We introduce **RSI-Forge**, a multi-agent pipeline that turns published papers into executable environments for self-improvement. Three agents coordinate construction, reproduction, and review to produce tasks with automated evaluators; each paper's method is independently reimplemented to establish a baseline score. We present 210 environments across 18 fields, including 90 reviewed by independent human domain experts. Both experts and agent judges give high ratings to the potential for improving the provided starting solutions and the evaluators' ability to distinguish solution quality, whereas experts are more critical of shortcut resistance, faithfulness to the source paper, and whether a single idea can exhaust a task. To validate their use for repeated improvement, we evaluate four models over 3 successive attempts on 120 environments, with each attempt inheriting prior code and notes while model weights remain fixed. At least one model improves after the first attempt in 84% of environments. Models also outperform the reproduced paper methods in 68 of the 120 environments, demonstrating room for gains beyond these baselines. Transcript analysis identifies work beyond parameter tuning in 95% of these successful attempts. Analysis of the resulting trajectories shows that models scoring lower on these tasks explore less, more often accept gains smaller than the reported standard error, and rely more heavily on tuning to the development set. **RSI-Forge** provides a scalable approach to constructing research environments for training and evaluating self-improving agents.

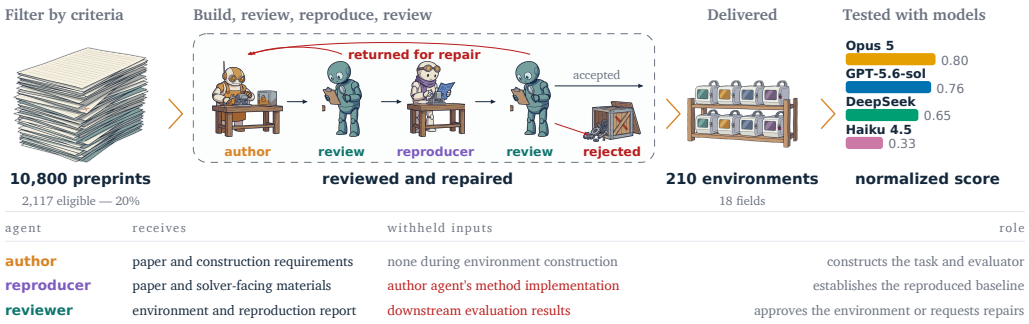
1 INTRODUCTION

Recursive self-improvement (RSI) is a process in which an agent improves its own capabilities and uses those improvements to guide further changes (Duan et al., 2026). Such agents could accumulate useful strategies and tackle increasingly difficult problems with less human intervention. Developing this capability requires environments that provide challenging problems and reliable feedback on attempted improvements. These environments supply experience for training and a basis for evaluation, making their availability and diversity central to the development of RSI at scale.

Constructing suitable RSI environments is particularly difficult for research tasks. They must pose meaningful problems, offer room for continued improvement, and evaluate solutions automatically while distinguishing genuine gains from measurement noise and scoring shortcuts. Formulating such tasks requires domain knowledge, and implementing reliable evaluators adds a separate engineering challenge. Existing benchmarks draw on repositories and competitions with established tests or graders (Jimenez et al., 2024; Chan et al., 2025), or rely on tasks and evaluation rubrics written by experts (Starace et al., 2025; Wijk et al., 2024). These approaches provide valuable environments, but their scale and domain coverage remain tied to the availability of evaluation resources and expert effort. This raises a central question: *how can we automatically construct RSI environments at scale?*

To address this question, we introduce **RSI-Forge**, a multi-agent pipeline that constructs executable RSI environments from published papers (Fig. 1). Papers provide concrete research questions, methods, and measured results that can guide environment design. An author agent develops the task,

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.



agent	receives	withheld inputs	role
author	paper and construction requirements	none during environment construction	constructs the task and evaluator
reproducer	paper and solver-facing materials	author agent's method implementation	establishes the reproduced baseline
reviewer	environment and reproduction report	downstream evaluation results	approves the environment or requests repairs

Figure 1: Overview of **RSI-Forge** and the information available to each agent. Failed checks lead to repair or rejection before an environment is accepted. The output count includes all 210 environments. The performance bars report median scores of final solutions after 3 successive attempts on 120 environments sampled from the main construction run. Scores are normalized so that the starting solution is 0 and the target score is 1 (Eq. (1)).

starting solution, and evaluator. A reproducer agent independently implements the paper's method using only the paper and solver-facing materials, establishing the reproduced baseline. A reviewer agent checks that the task can be understood from its instructions and that the evaluator measures the stated objective, requesting repairs when needed. The agents have distinct responsibilities and access to information, allowing reproduction and review to assess the environment separately from the author agent's construction process.

Using this pipeline, we construct 210 environments across 18 fields. To assess their quality, three agent judges evaluate the environments using a common rubric, and independent human domain experts review 90 environments within their fields of expertise. The rubric assesses whether tasks pose meaningful research problems, offer opportunities for improvement, and provide reliable evaluation. Experts assign their highest ratings to the potential for improving the starting solutions and the evaluators' ability to distinguish solution quality. They are more critical than agent judges of shortcut resistance, fidelity to the source paper, and whether a single idea can exhaust a task (Fig. 2).

To assess whether these environments support repeated improvement, we evaluate four solver models over 3 successive attempts on 120 of the environments. Each attempt inherits prior code and notes while model weights remain fixed. At least one model improves after the first attempt in 84% of environments, and the reproduced baseline is surpassed in 68 of the 120 environments. Transcript analysis identifies work beyond parameter tuning in 95% of these successful attempts. Analysis of the resulting trajectories further distinguishes how models explore alternatives, assess uncertain gains, and select solutions. Together, these results show that **RSI-Forge** can automatically construct research environments that support measurable improvement over successive attempts.

Our contributions are:

1. **An automated RSI environment construction pipeline.** We introduce RSI-Forge, which turns published papers into executable environments.
2. **A corpus spanning diverse research fields.** We construct 210 environments[1] across 18 fields, including 90 independently reviewed by human domain experts.
3. **Validation through repeated improvement.** We empirically demonstrate gains over successive attempts and beyond reproduced baselines, with trajectory analysis revealing differences in how models explore alternatives and assess changes.

2 Related Work

Self-improving agents. Prior work studies improvement at several levels. Self-Refine iteratively revises model outputs using feedback, while Reflexion retains reflections from earlier attempts to inform subsequent decisions (Madaan et al., 2023; Shinn et al., 2023). Other approaches optimize

[1]We provide 50 environments as examples in supplementary materials, and will also publicly release them.

prompts, workflows, agent designs, harnesses, or source code (Fernando et al., 2023; Zhang et al., 2024; Hu et al., 2024; Zelikman et al., 2024; Zhang et al., 2025; Yin et al., 2024; Lee et al., 2026). These approaches investigate how agents improve their solutions and procedures. Our work addresses the construction of environments that supply challenging problems and measurable feedback for such improvement.

Automated task and environment generation. Research on learning tasks spans curriculum learning, adaptive environment design, and autonomous task selection (Bengio et al., 2009; Dennis et al., 2020; Jiang et al., 2021; Wang et al., 2023). Absolute Zero and R-Zero generate tasks as part of a learning process that couples task proposal with solving (Zhao et al., 2025; Huang et al., 2025). AgentSynth synthesizes tasks and trajectories for computer-use agents (Xie et al., 2025), while EnvScaler and EnvFactory construct executable environments for tool interaction (Song et al., 2026; Xu et al., 2026). SPADE further couples environment generation with learning: a model designs executable environments and learns by acting within them (Liu et al., 2026). For terminal agents, TermiGen synthesizes verifiable environments through iterative multi-agent refinement (Zhu et al., 2026), while Fan et al. (2026) progressively increase environment difficulty to sustain learning signals. **RSI-Forge** shares the goal of expanding environment supply through automation, focusing on research problems drawn from published papers across diverse fields.

Research and engineering benchmarks. Existing benchmarks obtain tasks and evaluation procedures from several sources. SWE-bench uses repository issues and tests, while MLE-bench adapts machine-learning competitions (Jimenez et al., 2024; Chan et al., 2025). Related benchmarks assess machine-learning experimentation and computational reproducibility (Huang et al., 2024; Siegel et al., 2024). PaperBench evaluates paper replication using rubrics developed with the source papers' authors (Starace et al., 2025); RE-Bench provides expert-designed research engineering environments (Wijk et al., 2024); and Terminal-Bench provides curated terminal tasks with human-written solutions and verification tests (Merrill et al., 2026). RSI-Exam focuses specifically on repeated improvement in executable research tasks (Aiming Lab, 2026). Closer to automated construction from papers, Theiler et al. (2026) translate published methods into a shared benchmarking framework for machine health intelligence. **RSI-Forge** constructs research tasks and their evaluators across fields, with independent reproduction establishing baselines against which further improvements can be measured. Appendix A provides additional comparisons and discussion of verification.

3 RSI-FORGE

RSI-Forge uses published papers as a source of research problems and reference methods. It converts each selected problem into an executable environment and independently reimplements the paper's method within that environment to establish a measured baseline. A reviewer agent checks that the instructions describe the task clearly and that the evaluator measures the intended objective, requesting repairs when needed.

3.1 ENVIRONMENT FORMAT

We start by defining an RSI environment as an executable research task with the resources needed to develop and evaluate solutions. Each environment specifies a research objective, the required solution interface, and a compute budget. It provides a runnable starting solution, code for generating task data, and an automated evaluator. We refer to the agent attempting the task as the solver. The solver can modify or replace the starting solution and test its changes on a development set. A separate held-out test set is reserved for grading submitted solutions.

Each environment includes a reproduced baseline: the source paper's method independently reimplemented and evaluated under the environment's task definition and resource limits. Its measured score provides a reference for assessing improvements beyond that method. The source paper, reproduced implementation, and test data are not supplied as solver inputs.

Each environment also defines a target score based on a task-specific performance bound or an idealized solution, such as one with access to information unavailable to the solver. The target serves as a normalization reference and need not be attainable with the information and resources available to the solver. To compare performance across environments, we normalize scores so that the starting solution corresponds to 0 and the target to 1.

3.2 Multi-Agent Construction

Construction begins with papers whose research problems can be evaluated automatically and whose methods fit the available compute budget. For each paper, an author agent adapts the research problem into an executable task, specifying the solver's objective, permitted changes, and resource limits. It builds the starting solution, code for generating task data, and evaluator, and prepares the development and held-out test sets. The author agent also implements the paper's method for preliminary checks and measures the starting-solution and target scores.

The proposed environment then undergoes automated checks and review. A reviewer agent, with a context separate from the author agent's, examines the instructions, implementation, and check results to assess whether the task can be understood from the supplied materials and whether the evaluator measures the intended research objective. Initial checks verify that the components run and that the evaluator executes the code submitted through the required interface. Scripted tests also check whether performance differences exceed measured variation across repeated evaluations. The author agent specifies how the test cases differ from the development cases—for example, by reserving different simulation settings for testing—and the reviewer agent checks this distinction.

Once these checks pass, a reproducer agent receives the paper together with the task instructions, development data, and evaluator available to a solver. The author agent's implementation of the method, construction history, the test data, and the target score are withheld. The reproducer agent independently implements the paper's method using the same submission interface and compute budget as solver solutions. Its implementation is evaluated on the held-out test set to establish the reproduced baseline score. The reproduction must outperform the starting solution for the environment to pass this stage, providing direct evidence that improvement over the starting solution is possible.

The reviewer agent then examines the reproduction report, and a final end-to-end test checks that the environment can start, accept a solver's submission, and grade it. Acceptance requires successful automated checks and approval from the reviewer agent at every stage. A failed check or unresolved review finding returns the environment to the author agent for repair, and environments that still fail after the permitted repair attempts are rejected.

3.3 Scoring and Verification

Scoring solver submissions. Each environment measures its research objective using a scoring program that is deterministic for fixed evaluation inputs. To enable performance comparison across environments, we normalize a raw test score v using the starting-solution score w and target score u, both established on the same test set:

$$b(v) = \frac{v - w}{u - w}. \tag{1}$$

This maps the starting solution to 0 and the target to 1, with the reproduced baseline providing a separate comparison point on this scale.

Submission validity. Alongside performance scoring, automated checks assess whether submitted solutions satisfy the required file structure, interfaces, correctness conditions, and resource limits. A model judge agent, distinct from the reviewer agent used during environment construction, also examines the submitted code for violations of task-specific rules. It assigns zero reward to a violating submission regardless of its measured performance.

Preventing reward hacking. Agents may exploit access to evaluation data or tests to increase their scores without improving their solutions (Zhong et al., 2025; Prasad et al., 2026). To mitigate such risk, **RSI-Forge** restricts access to held-out grading materials. Before submitted code runs, the evaluator loads the test data into memory and removes their on-disk copy. The solver can use development evaluations while working, but held-out scores are not returned during solving. These restrictions do not eliminate all opportunities for exploitation: some environments left other grading materials accessible. We report the observed access failures and the sensitivity of solver results to those failures in Section 5 and Appendix D.4.6.

Table 1: Main-run construction funnel. Counts include environments that passed after repairs. The additional 30 environments built with GPT-5.6-sol are outside this funnel.

Stage	Count
Source papers screened	10,800
Papers eligible for construction	2,117
Construction attempts begun	213
Passed Design	199
Passed Build	198
Passed Independent Reproduction	189
Passed End-to-End Testing	180
Included in the main corpus	180

Table 2: Examples of environments from the main construction run.

Field	Task	Evaluation objective
Robotics	Choose footholds on a terrain map	Increase distance in six steps
Life Sciences	Propose molecules for a binding pocket	Improve docking scores
Earth, Climate & Space	Estimate weather from partial observations	Reduce state-estimation error
Statistics & Inference	Estimate effects without a control series	Reduce treatment-effect error
Software Engineering & Formal Methods	Generate tests within a fixed budget	Trigger more distinct failures
Computational Physics	Infer a plasma boundary from magnetic data	Reduce boundary-estimation error

4 Building the Corpus

4.1 Sourcing

We collect recent preprints separately by domain, then deduplicate the resulting list to broaden disciplinary coverage. Each paper is read in full against six eligibility criteria: a quantitative objective with a stated direction of improvement, a measured baseline, a metric that can be computed by a deterministic script, a method that fits the available compute budget, measurable room for improvement, and a licence permitting redistribution. All source papers are from 2026. These criteria determine which papers enter construction; they do not guarantee that an environment can be built. Appendix B.1 details the criteria, and Section 4.2 reports the counts.

4.2 Construction Funnel

The main construction run used Claude Opus 5 for the author agent, reproducer agent, and reviewer agent, with separate sessions and role-specific inputs. Table 1 traces the source papers through eligibility and construction. We stopped after 180 accepted environments; only 213 of the 2,117 eligible papers entered this run, so the counts do not estimate the yield from all eligible papers.

Among accepted environments, 79% required at least one revision after failing a construction check. The funnel therefore reports passage after any permitted repairs. Analysis of repair records identified recurring defects in task instructions and evaluation, summarized in Table 3. Construction used a median of 170 million model tokens per accepted environment. Across construction, 99.6% of input tokens were cache reads. Appendices B.4 and C.1 provide the repair breakdown and construction costs.

We then replaced the construction model with GPT-5.6-sol and built 30 additional environments using the same queue and checks. This yields 210 environments overall. The two construction sets are unpaired by source paper and field; the second run tests whether the procedure works with another backend, rather than estimating a paired model effect (Appendix D.5).

Table 3: Most common construction defects identified by model-based coding of repair passages. Percentages are shares of repair passages; categories overlap. Table 6 gives the full breakdown.

Issue identified	Share of repair passages
Checks unable to detect the intended failure	66%
Incorrect or incomplete instructions	63%
Checks inspecting a different copy of the code	54%
Score gains without performing the intended work	37%

4.3 Corpus Composition and Contents

The full corpus contains 210 environments across 18 research fields. The 180 environments from the main construction run include 6–14 environments per field. Coverage extends beyond machine learning to areas including robotics, computational physics, statistics, and software engineering. Table 2 showcases example tasks and evaluation objectives from six research fields. Table 7 reports field counts, example tasks, and objectives; Appendix C.3 provides detailed examples.

For each environment, we retain the task instructions, runnable code, code for generating task data, starting solution, reproduced baseline implementation, and evaluation configuration. Environments provide a development set for experimentation and a held-out test set for scoring. Construction records document feedback from the reviewer agent and repairs.

The corpus is accompanied by quality assessments and records of agent experiments. These include human and automated ratings with written justifications, agent transcripts, saved programs, experiment logs, and held-out scores. Together, they support inspection of both the constructed environments and the sequence of changes agents make while solving them.

Example environment. In one weather-estimation environment, the solver estimates a state from a small ensemble and noisy observations of half its components (Table 12). The starting solution uses an ensemble Kalman method based on the raw sample covariance. The solver can test changes on 160 development instances, while grading uses 324 held-out instances generated under different simulation conditions. The objective is to minimize mean estimation error. On the held-out test set, the starting solution has an error of 1.786 and the independently reproduced paper method achieves 1.390. With a target score of 1.156, normalization maps the starting solution to 0 and the reproduced baseline to approximately 0.629 (Eq. (1)).

4.4 Quality

Three agent judges rated all 180 main-run environments, and human domain experts rated 90. Both groups used the same nine-dimension rubric, with each dimension rated from 1 to 5. On the expert-reviewed subset, the means across environments and rubric dimensions were 4.24 and 4.54 for human experts and the three-judge panel, respectively. Experts rated room above the starting solution (4.81) and discrimination between attempts (4.74) highest. These ratings support the environments' core measurement properties (Fig. 2 and Table 9).

Experts were more critical of saturation, shortcut resistance, and fidelity to the paper. The widest difference concerned whether one idea could exhaust a task: experts assigned 3.09, compared with 3.98 from the agent panel. This comparison also reflects unequal information: some experts cited the solver-result summaries withheld from agent judges. Fidelity disagreements often concerned simplifications made to fit the compute budget. Written justifications and rater correlations are reported in Appendix C.2.1.

Removing the most generous judge lowers the panel mean to 4.39 without eliminating the largest disagreements. For the additional 30 environments built with GPT-5.6-sol, all three judges give mean ratings within a fifth of a point of their main-run ratings (Appendix D.5). Under the same rubric, agent judges rated our environments higher than manually constructed RSI-Exam tasks on distinguishing solution quality and resisting shortcuts, whereas RSI-Exam received higher ratings on resistance to saturation (Table 10). Review ratings are not used to select or weight solver outcomes.

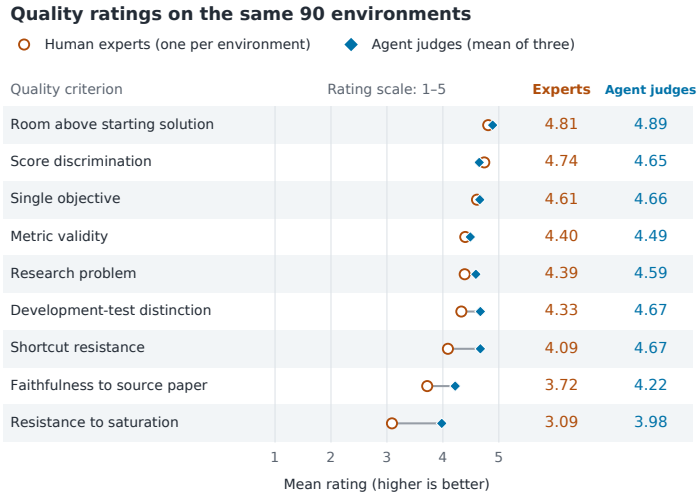


Figure 2: Quality ratings from human experts and agent judges on the same 90 environments, with one expert and three judges per environment. Points show means on a 1–5 scale (higher is better). Agent ratings are averaged within environments; rows are ordered by expert mean. Only experts received available solver results. Appendix C.2.2 and Table 9 give the rubric and per-dimension standard deviations.

5 Experiments

We use the environments to measure improvement over successive attempts and relative to the reproduced baselines, and analyze how agents explore alternatives, assess changes, and select solutions.

5.1 Experimental Setup

Before inspecting scores, we randomly sampled 120 environments from the main construction run, covering all 18 fields. Four solver agents use the same Claude Code CLI harness, with only the model endpoint changed: Claude Opus 5, GPT-5.6-sol, DeepSeek-V4-Flash, and Claude Haiku 4.5.

Each model runs 3 successive sessions per environment, yielding 480 model–environment chains and 360 sessions per model. Each session begins with a fresh context. The first receives the starting solution; subsequent sessions inherit the previous session's submitted solution, saved versions, experiment log, and notes. A session ends when the agent stops or after 60 minutes, whichever comes first. Almost all sessions ended when the agent stopped. This session budget is separate from each environment's budget for running a candidate method. Models do not exchange records in the main experiment, and their weights remain fixed.

We report held-out scores of submitted solutions after each session, using the final submission after session 3 as the primary outcome. We also replay every saved intermediate version on the held-out test set to measure progress within sessions and assess whether agents submit their best saved solutions. Comparisons use normalized scores and within-environment ranks; bootstrap intervals resample environments. Appendix D.2 gives the full protocol, and Tables 14 and 15 report versions, turns, tool calls, and output tokens.

5.2 Performance and Continued Improvement

Table 4 reports submitted scores after each session. The two strongest models have overlapping median-score intervals; both outperform DeepSeek-V4-Flash, which outperforms Claude Haiku 4.5. Within-environment ranks give the same ordering, and paired bootstrap comparisons separate five of the six model pairs. These results describe performance on this corpus under the common harness. The same solver ordering holds on the 30 environments constructed with GPT-5.6-sol, although the two construction sets are unpaired by source paper and field (Table 19).

Table 4: Submitted-solution scores on 120 environments, normalized to the starting solution (0) and target (1). Session 3 confidence intervals use bootstrap resampling of environments. Mean rank averages within-environment ranks at session 3 (1 is best).

	Median normalized score ↑			Session 3	
Model	Session 1	Session 2	Session 3	95% CI	Mean rank ↓
Claude Opus 5	0.754	0.791	0.800	[0.753, 0.867]	1.67
GPT-5.6-sol	0.673	0.747	0.761	[0.723, 0.824]	1.73
DeepSeek-V4-Flash	0.550	0.594	0.647	[0.586, 0.711]	2.97
Claude Haiku 4.5	0.129	0.277	0.332	[0.203, 0.451]	3.63

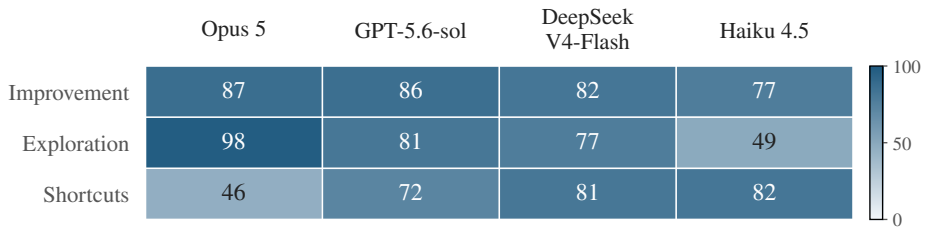


Figure 3: Transcript-coded behavior across 360 sessions per model (120 environments, three sessions each). Cell values and color intensity indicate the percentage of sessions assigned to each group by model-based open coding. Groups overlap; "improvement" covers practices for developing and checking solutions, without requiring score gains.

Most score improvement occurs in the first session for the three stronger models, with smaller gains in later sessions. With a 0.01 threshold on gains in normalized score, at least one model improves after the first session in 84% of environments. In 68 of the 120 environments, at least one model surpasses the reproduced baseline in one of its 3 sessions. These comparisons establish progress within the constructed tasks and against their reproduced baselines (Appendix D.3).

Final solution selection. Some final submissions score below an earlier saved version from the same agent, even when the better version passes the automated submission checks. Replaying intermediate versions exposes this difference between discovering a solution and selecting it.

Access to held-out material. The transcript coding records access to held-out material or another run's graded record in 22 of 480 chains. We retain these chains in the main results. Excluding them changes each model's median normalized score by at most 0.006 and preserves the ordering (Appendix D.4.6).

5.3 Transcript Analysis

We analyze the records to characterize how agents work. A model-based coding procedure groups practices into *improvement* (practices for developing and checking solutions), *exploration* (tests of whether a change is useful), and *shortcuts* (score-directed behavior without sufficient evidence of underlying improvement). For the analysis shown in Fig. 3, Claude Opus 5 identifies recurring practices without a predefined category list. Each positive assignment requires supporting quotations from the transcript. Humans did not label these trajectories (Appendices D.4 and D.4.6).

Figure 3 summarizes the prevalence of the three behavior groups across models. All four models frequently modify their solutions, but their evidence-gathering practices differ. Exploration is more common for Claude Opus 5 than for Claude Haiku 4.5, while fitting to the development score follows the reverse pattern. The middle models exchange positions on individual practices.

We check these patterns using a separate analysis with a fixed codebook of ten practices. Three additional model coders apply this codebook to a stratified sample of 30 sessions per solver model, obtaining agreement of $\kappa = 0.73$ with the original coder and recovering the same contrast in exploration and shortcuts between Claude Opus 5 and Claude Haiku 4.5 (Appendix D.4.6). A further check uses numeric logs from 139 sessions with recoverable scores and standard errors. Gains

smaller than the reported standard error account for 73% of recorded accepted improvements for Claude Haiku 4.5, compared with 44% for Claude Opus 5 (Appendix D.4.5).

Among sessions that outperform the reproduced baseline, 95% are coded as involving work beyond parameter tuning; 5% are tuning-only. This coding characterizes the work performed in these tasks without establishing novelty to the source paper's field. Table 16 and Fig. 7 report the categories and their frequencies, and Appendix D.3 gives examples of method changes.

6 Discussion

Our results show that published papers can serve as a practical source of executable research environments across diverse disciplines. Most accepted environments required repairs before passing the construction checks (Section 4.2). The recurring defects included incorrect instructions and checks that could miss failures, showing that verification and repair are substantial parts of automated environment construction.

The quality reviews clarify what this construction process achieves. Experts rated room above the starting solution and the evaluator's ability to distinguish attempts most highly, supporting the properties needed to measure progress. Their lower ratings for resistance to saturation and fidelity to the source paper identify a harder challenge: preserving the depth of a research problem when making it executable under a fixed budget. The disagreement with automated judges suggests that targeted expert review remains valuable for assessing these properties as construction scales.

The solver experiments demonstrate both improvement over successive attempts and gains beyond reproduced baselines. Gains are nevertheless concentrated in the first session for the stronger models. Diminishing gains do not by themselves establish task saturation; they may also reflect limits in how agents search for alternatives or use inherited records.

The trajectories also reveal weaknesses in how agents conduct experiments and retain their results. Lower-scoring models explored less and more often accepted gains smaller than the reported standard error. Across models, some final submissions were worse than previously saved versions, showing that finding a better solution does not ensure it is retained. These observations make experimental judgment and final selection concrete targets for future interventions. The saved programs, measurements, and handover records provide a basis for evaluating such interventions; the present associations do not establish which practices cause higher performance.

Limitations. Tasks are adapted to bounded compute, so surpassing a reproduced baseline does not establish an advance on the original research problem. Our sessions measure progress on individual tasks, and the study does not isolate the effect of inherited records from additional solving time. No trajectories were used for training, and the study does not establish broader gains in research capability. Some environments left held-out grading materials accessible during solver runs. Expert review covers 90 of 180 main-run environments, with one expert each and no human inter-rater estimate; trajectory labels are model-generated without human validation. We do not ablate construction roles, so their individual contributions remain unresolved. Access restrictions also limit independent replication of the full corpus (Appendix E.3).

7 Conclusion

We introduced **RSI-Forge**, a multi-agent pipeline that turns published papers into executable RSI environments with automated evaluators and independently reproduced baselines. The resulting corpus contains 210 environments across 18 fields. Reviews by human experts highlight opportunities to improve the starting solutions and the evaluators' ability to distinguish solution quality, while identifying challenges in shortcut resistance and fidelity to the source paper. Solver experiments demonstrate gains over successive attempts and beyond reproduced baselines. The trajectories further reveal differences in how agents explore alternatives, assess uncertain gains, and select solutions. **RSI-Forge** provides a scalable approach to expanding the supply of research environments for developing and evaluating self-improving agents.

AI USE STATEMENT

Language models are used throughout the construction and evaluation procedures reported in this paper. They construct environments, independently reproduce source methods, review environments, assess submission validity and environment quality, and generate solver trajectories (Sections 3 to 5). They also analyze construction records and solver transcripts (Appendices B.5 and D.4). The corresponding sections and appendices describe the models, inputs, and evaluation procedures.

We also used language models to identify relevant literature, revise the manuscript, refactor analysis and figure code, create Fig. 1, and review methodological descriptions and interpretations of results. The authors reviewed the resulting manuscript, analyses, and figures and take responsibility for the paper's content, including AI-assisted text, code, and visual material.

REPRODUCIBILITY STATEMENT

We describe environment construction and acceptance checks in Section 3 and Appendix B, quality-review procedures and the rubric in Appendices C.2 and C.2.2, and the solver protocol and transcript coding in Section 5.1 and Appendices D.2 and D.4. The supplementary materials provide 50 environments, including ratings and written justifications from independent human domain experts for 25 of them. The environment packages include task instructions, scoring code, and evaluation configurations. Some data and container-build files are omitted to meet the upload size limit; these are documented in the supplement and must be restored to execute or grade affected environments. Repeating these procedures also requires access to the model versions used.

REFERENCES

Aiming Lab. RSI-Exam: Benchmarking recursive self-improvement through executable research. https://rsi-exam.ai/, 2026.

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *International Conference on Machine Learning*, 2009.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. In *International Conference on Learning Representations*, 2025.

Mingguang Chen, Licheng Wang, and Bo Qu. Recursive self-improvement in ai: From bounded self-refinement to autonomous research loops. *arXiv preprint arXiv:2607.07663*, 2026.

Michael Dennis, Natasha Jaques, Eugene Vinitsky, Alexandre Bayen, Stuart Russell, Andrew Critch, and Sergey Levine. Emergent complexity and zero-shot transfer via unsupervised environment design. In *Advances in Neural Information Processing Systems*, 2020.

Yi Duan, Ying Liu, Zirui Tang, Haodong Chen, Jun Zhou, Yumou Liu, Bangrui Xu, Yukai Wu, Sidi Chen, Yuhan Zhou, Haoyu Wang, Xiaoyou Yu, Shaokun Han, Xuzhou Zhu, Le Zhou, Bolin Lu, Wei Zhou, Jiachen Liu, Nuozhou Fang, Jiaxin Tian, Ruoyu Chen, Yuxuan Li, Kai Zuo, Kaiyan Zhang, Qianyu Yang, Zijie Wang, Jiantao Qiu, Conghui He, Guoliang Li, Bowen Zhou, Zhiyuan Liu, Zhoufutu Wen, Jihua Kang, Xuanhe Zhou, and Fan Wu. The last AI built by humans: Toward genuine recursive self-improvement, 2026.

Zhiyuan Fan, Tinghao Yu, Yuanjun Cai, Jiang Zhou, Jiangtao Guan, Jincheng Liu, Yun Yang, Dingxin Hu, Zhuo Han, Xing Wu, Feng Zhang, and Lilin Wang. Environment evolution for terminal agents, 2026.

Chrisantha Fernando, Dylan Banarse, Henryk Michalewski, Simon Osindero, and Tim Rocktäschel. Promptbreeder: Self-referential self-improvement via prompt evolution, 2023.

Odd Erik Gundersen and Sigbjørn Kjensmo. State of the art: Reproducibility in artificial intelligence. In *AAAI Conference on Artificial Intelligence*, 2018.

Diandian Guo, Cong Cao, Fangfang Yuan, Yingqi Wang, Yueshan Wang, and Dakui Wang. Self-authored verification is unreliable in heuristic self-improving agents. *arXiv preprint arXiv:2607.24300*, 2026.

Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *AAAI Conference on Artificial Intelligence*, 2018.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems, 2024.

Chengsong Huang, Wenhao Yu, Xiaoyang Wang, Hongming Zhang, Zongxia Li, Ruosen Li, Jiaxin Huang, Haitao Mi, and Dong Yu. R-Zero: Self-evolving reasoning LLM from zero data, 2025.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. MLAgentBench: Evaluating language agents on machine learning experimentation. In *International Conference on Machine Learning*, 2024.

Minqi Jiang, Edward Grefenstette, and Tim Rocktäschel. Prioritized level replay. In *International Conference on Machine Learning*, 2021.

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*, 2024.

Douwe Kiela, Max Bartolo, Yixin Nie, Divyansh Kaushik, Atticus Geiger, et al. Dynabench: Rethinking benchmarking in NLP. In *Conference of the North American Chapter of the Association for Computational Linguistics*, 2021.

Hyunin Lee, Jinglue Xu, Jeffrey Seely, Donghyun Lee, Matei Zaharia, and Yujin Tang. Recursive harness self-improvement, 2026.

Bo Liu, Simon Yu, Yiding Jiang, Ao Qu, Andrew Zhao, Zichen Liu, Junsu Kim, Zijian Zhou, Seungone Kim, Tongzheng Ren, Mickel Liu, Hanfei Yu, Zhaorun Chen, Weiyan Shi, Paul Pu Liang, Luke Zettlemoyer, Yejin Choi, and Natasha Jaques. SPADE: Self-play in adaptive synthetic executable environments, 2026.

Yaxin Luo, Haobin Jiang, Jialv Zou, Xu Huang, Wenhao Yan, Haodong Li, et al. Autodesign: Meta-harness optimization for long-horizon agentic design. *arXiv preprint arXiv:2608.13560*, 2026.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, et al. Eureka: Human-level reward design via coding large language models. In *International Conference on Learning Representations*, 2024.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, et al. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023.

Anas Mahmoud, MohammadHossein Rezaei, Mingchen Zhuge, Chenguang Wang, Yuandong Tian, and Yunzhong He. Announcing RSI Bench. https://labs.scale.com/rsi-benchmark/blog/announcing-rsi-bench, 2026. Scale AI. Announced 7 August 2026.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

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

Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, et al. Improving reproducibility in machine learning research (a report from the NeurIPS 2019 reproducibility program). *Journal of Machine Learning Research*, 2021.

Pradyumna Shyama Prasad, Meiri Anto, Leon Eshuijs, Julian Moncarz, Kaustubh Kislay, and Juan J. Vazquez. BAITBENCH: Measuring agent reward hacking with optional shortcuts planted in ML tasks. *arXiv preprint arXiv:2608.30724*, 2026.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, et al. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *International Conference on Learning Representations*, 2024.

Amit Roth, Ivan Bercovich, and Yonathan Efroni. Hack-verifiable terminal bench: Evaluating reward hacking in terminal tasks. *arXiv preprint arXiv:2608.22103*, 2026.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.

Zachary S. Siegel, Sayash Kapoor, Nitya Nadgir, Benedikt Stroebl, and Arvind Narayanan. CORE-Bench: Fostering the credibility of published research through a computational reproducibility agent benchmark, 2024.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. EnvScaler: Scaling tool-interactive environments for LLM agent via programmatic synthesis. In *Findings of the Association for Computational Linguistics: ACL 2026*, 2026. doi: 10.18653/v1/2026.findings-acl.407. URL https://aclanthology.org/2026.findings-acl.407/.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, et al. PaperBench: Evaluating AI's ability to replicate AI research, 2025.

Raffael Theiler, Ludovico Comito, David Leko, Leandro Von Krannichfeldt, Lev Telyatnikov, and Olga Fink. From paper to benchmark: agentic, framework-based reproduction of under-specified methods in machine health intelligence. *arXiv preprint arXiv:2605.28371*, 2026.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, et al. Voyager: An open-ended embodied agent with large language models, 2023.

Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, et al. Large language models are not fair evaluators. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

Zefeng Wang, Minxi Yan, Jinhe Bi, Sikuan Yan, Volker Tresp, and Yunpu Ma. Metaskill-evolve: Recursive self-improvement of llm agents via two-timescale meta-skill evolution. *arXiv preprint arXiv:2607.05297*, 2026.

Hjalmar Wijk, Tao Lin, Joel Becker, Sami Jawhar, Neev Parikh, et al. RE-Bench: Evaluating frontier AI R&D capabilities of language model agents against human experts, 2024.

Yuhao Wu, Jingyuan Zhang, Jiajun Shi, Xinping Lei, Qingshui Gu, Yuxuan Zhang, et al. Harness-dev: Can llms create and evolve their own agent harness? *arXiv preprint arXiv:2609.01437*, 2026.

Jingxu Xie, Dylan Xu, Xuandong Zhao, and Dawn Song. AgentSynth: Scalable task generation for generalist computer-use agents, 2025.

Minrui Xu, Zilin Wang, Mengyi DENG, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, Heyuan Deng, Fei Mi, Lifeng Shang, Xingshan Zeng, and Zhijiang Guo. EnvFactory: Scaling tool-use agents via executable environments synthesis and robust RL, 2026.

Shuhan Xue, Zixin Ding, Yichen Shen, Yinjie Wang, Zhenfei Yin, Yingcheng Wu, Yuxin Chen, Mengdi Wang, and Ling Yang. PAST-Bench: Benchmarking the foundations of recursive self-improvement in personal agents, 2026.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursive self-improvement, 2024.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling (COLM)*, 2024.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents, 2025.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation, 2024.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data, 2025.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*, 2023.

Ziqian Zhong, Aditi Raghunathan, and Nicholas Carlini. ImpossibleBench: Measuring LLMs' propensity of exploiting test cases. *arXiv preprint arXiv:2510.20270*, 2025.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, Emad Barsoum, William Yang Wang, and Wenbo Guo. TermiGen: High-fidelity environment and robust trajectory synthesis for terminal agents, 2026.

A Extended Related Work

What changes during self-improvement. Beyond output refinement and agent-design optimization (Section 2), recent work studies changes to procedural skills, agent harnesses, and the improvement procedure itself (Chen et al., 2026; Wang et al., 2026; Wu et al., 2026; Luo et al., 2026). Our experiments hold model weights and the solver harness fixed and measure changes to task solutions across successive sessions.

Automating task and environment components. Related approaches generate task components within existing systems: Voyager proposes goals and develops reusable skills (Wang et al., 2023); Eureka generates reward code (Ma et al., 2024); and ToolLLM synthesizes tool-use instructions and solution paths (Qin et al., 2024). Dynabench uses human contributions to construct challenging evaluation examples (Kiela et al., 2021).

Benchmark construction and human input. Table 5 compares selected research and engineering benchmarks by evaluation procedure and human contribution. RSI-Exam provides manually contributed and reviewed research tasks (Aiming Lab, 2026). RSI Bench evaluates skills involved in frontier AI development (Mahmoud et al., 2026), while PAST-Bench tests whether retained experience improves personal agents across sessions (Xue et al., 2026).

Evaluation independence and reproducibility. Model judges can favor their own outputs and exhibit other biases, including sensitivity to answer order and verbosity (Panickssery et al., 2024; Zheng et al., 2023; Wang et al., 2024). Related studies examine discrepancies between self-authored tests and external evaluations (Guo et al., 2026), and reward hacking that satisfies task checks without fulfilling the intended objective (Roth et al., 2026). These concerns motivate separating solution development from final evaluation and comparing agent judgments with human assessments. In our main construction run, the three agent roles use the same underlying model in separate contexts (Section 4.2 and Appendix B.2); this separation does not establish that their errors are independent.

Work on reproducibility emphasizes access to code and documented experimental procedures (Gundersen and Kjensmo, 2018; Pineau et al., 2021). In reinforcement learning, variation across runs can also make apparent improvements difficult to interpret (Henderson et al., 2018). These concerns

Table 5: Selected research and engineering benchmarks, compared by evaluation procedure and human contribution. Size denotes tasks or environments (papers to replicate for PaperBench), using the cited versions. The **RSI-Forge** count includes both construction runs.

System	Evaluation	Human contribution	Size
SWE-bench (Jimenez et al., 2024)	Repository tests	Task curation; existing tests	2,294
MLE-bench (Chan et al., 2025)	Local implementations of competition metrics	Competition curation and grading setup	75
PaperBench (Starace et al., 2025)	Rubrics applied by model judges	Rubrics co-developed with the source papers' authors	20
RE-Bench (Wijk et al., 2024)	Task-specific automated scorers	Environment and scorer design	7
Terminal-Bench 2.0 (Merrill et al., 2026)	Task-specific tests	Task curation, reference solutions, and tests	89
RSI-Forge	Automated evaluator and independently reproduced baseline	Automated construction; 90 environments reviewed by human experts	210

inform our use of executable evaluators, independently reproduced baselines, and checks of measurement variation. The limitations of access to our corpus and construction implementation are described in Appendix E.3.

B Construction Details

B.1 Source Selection

We collect recent preprints separately by domain and deduplicate the combined list to broaden disciplinary coverage. The resulting set contains 10,800 papers. Each paper is read in full and assessed against six criteria. The first five concern the feasibility of constructing a task with measurable improvement; the sixth concerns redistribution.

Quantitative objective The research problem must have a quantitative objective with a stated direction of improvement. Additional quantities may serve as fixed constraints; unconstrained trade-offs between objectives do not provide the single ranking required by our format.

Reported baseline The paper must report measured performance on the objective, supported by experiments or datasets. This guides construction of the reproduced baseline; the reported value does not replace measurement within the adapted environment.

Automated evaluation A deterministic scoring program must be able to compute the objective from fixed evaluation inputs, without requiring a human to grade each solution.

Compute feasibility The source method must fit the compute budget available for running candidate solutions.

Room for improvement The problem must offer scope for improvements that can be distinguished from evaluation noise. This screening criterion does not establish that the environment's eventual target score is attainable.

Redistributable licence The source paper's licence must permit the required redistribution. This criterion can exclude papers that satisfy all five task criteria.

Of the 10,800 papers, 2,117 pass all six criteria. As described in Section 4.2, only a subset enters the main construction run. We do not release the sourcing procedure, per-paper verdicts, or rejection reasons, to avoid attaching our assessments of benchmark suitability to individual preprints and their authors.

B.2 Agent Roles and Information Access

Construction uses three agent roles with separate contexts and role-specific inputs. The main run uses Claude Opus 5 for all three roles (Section 4.2). These construction agents are distinct from the solvers and the agent judges used to assess environment quality.

Author agent. The author agent receives the source paper and construction requirements. It creates the task instructions, starting solution, data-generation code, and evaluator, and implements the

source method for preliminary checks. It revises the environment in response to failed checks and feedback from the reviewer agent.

Reviewer agent. The reviewer agent examines the task instructions, implementation, and check results to assess whether the supplied materials define a usable task and whether evaluation measures the intended objective. It reviews the reproduction report and provides acceptance decisions or requests for repair.

Reproducer agent. The reproducer agent receives the source paper and the task instructions, development data, and evaluator available to a solver. The author agent's method implementation and construction history, held-out test data, and target score are withheld. The reproducer agent implements the source method using the same submission interface and compute budget as solver solutions. A separate grading step evaluates this implementation on the held-out test set to establish the reproduced baseline score. The environment cannot pass reproduction unless this score improves on the starting solution.

B.3 Acceptance Checks

Every construction stage requires both successful automated checks and approval from the reviewer agent. The four stages below correspond to the construction funnel in Table 1; the implementation identifiers are given in parentheses.

Design (`design`) The data generator, evaluator, and author agent's implementation of the source method must exist and run. Checks assess whether performance differences exceed measured variation across repeated evaluations. The development and test sets must differ along a documented task condition, rather than solely through a random partition.

Build (`build`) The environment must load with all required files present, and the evaluator must read the solution submitted through the specified interface. Execution must produce the expected completion indicators, and the options used to replay solutions must match the evaluator's interface.

Independent reproduction (`reproduce`) The reproducer agent's implementation is evaluated through the same execution path and container as solver solutions. Its score must improve on the starting solution's score.

End-to-end testing (`smoke`) A solver container must complete startup, submission, and grading. Before submitted code runs, the evaluator loads the held-out test data into memory and removes their on-disk copy. These checks do not establish that all grading materials are inaccessible; observed access failures are reported in Section 5.2 and Appendix D.4.6.

A failed check or unresolved review finding returns the environment to the author agent. At each stage, the author agent has at most 3 repair attempts before rejection. Evaluation records distinguish a missing result from a measured score of zero.

B.4 Repair Statistics

Among the 180 accepted main-run environments, 79% required at least one repair: 53% during design, 45% during build, 21% during independent reproduction, and 20% during end-to-end testing. Repairs at one or both of the latter two stages occurred in 35% of accepted environments. The stage shares overlap because an environment may be repaired at multiple stages. These statistics concern accepted environments; Table 1 reports the full main-run construction funnel.

The author agent makes repairs within its original construction session. The reviewer and reproducer agents retain their respective session records. Across construction, 99.6% of input tokens are cache reads. Appendix C.1 reports token use, and Table 6 describes defects identified in the repair records.

B.5 Construction Defects and Review Topics

We use model-based coding to summarize two parts of the construction record. The first analysis assigns defect categories to repair passages written by the author agent. The second records topics addressed in the reviewer agent's acceptance verdicts, whether or not a verdict identifies a defect.

Table 6 reports the share of passages or verdicts assigned to each category. These are not percentages of environments, and a passage or verdict may receive multiple labels.

Table 6: Model-coded construction defects and review topics. Shares use repair passages as the denominator in the first block and reviewer-agent verdicts in the second; categories overlap within each block. A review topic does not by itself establish a defect.

Defects identified in repair passages	
Category	Share
Checks unable to detect the intended failure	66%
Incorrect or incomplete instructions	63%
Checks inspecting a different copy of the code	54%
Score gains without performing the intended work	37%
Checks that pass only for a particular phrasing	30%
Grading path not exercised during construction	29%
Target score nearly reached by a trivial solution	28%
Judgment-based scoring without a task measurement	21%
Metric unable to distinguish between solutions	18%
Provided materials allow reconstruction of the held-out test set	9%
Other	2%
Topics addressed in reviewer-agent verdicts	
Topic	Share
Instructions sufficient for planning a solution	75%
Checks' ability to detect the intended failures	74%
Room for improvement above the starting solution	62%
Independent reproduction of the baseline score	44%
Resistance to unauthorized access to test data	42%
Distinguishing outcomes in evaluation records	41%
Meaningful differences between development and test cases	39%
Actionable feedback for repairs	32%
Comparison with simpler methods	29%
Attribution of failures	26%

C Corpus Details

C.1 Construction Cost and Field Coverage

Table 1 reports the main construction funnel, and Appendix B.4 reports repair rates. Across accepted main-run environments, construction uses a median of 170 million model tokens. The median per-environment token shares are 82% for the author agent, 14% for the reviewer agent, and 3% for the reproducer agent. These shares are summarized separately across environments and therefore need not sum to 100%.

Table 7 gives field counts and example tasks for the 180 main-run environments. The additional 30 environments constructed with a second backend are described in Appendix D.5.

C.2 Review Protocol and Results

Three agent judges rated all 180 main-run environments, and independent human domain experts rated 90 of them. Both groups used the same 9-dimension rubric, with scores from 1 to 5 and written descriptions of the rating levels (Table 11).

Agent judges. Each judge uses its vendor's harness: Claude Opus 5 under Claude Code, GPT-5.6-sol under Codex at high reasoning effort, and Gemini 3.7 Flash under Antigravity. Each model–harness pair constitutes one agent judge. The judges read the task instructions and evaluator and can run the code, with network access for field knowledge. The author agent's construction transcript, held-out test data, and solver-result summaries are withheld.

Human expert reviews. Each expert holds a doctorate in the relevant field and is independent of environment construction. One expert reviews each environment and provides a written justifica-

Table 7: Field composition of the 180 main-run environments. Each row describes one example task from that field; arrows indicate the direction of improvement in its raw metric.

Field	Count	Example task	Evaluation objective
Robotics	14	A humanoid chooses footholds from a local terrain map.	Increase the distance reachable in six footsteps. ↑
Life Sciences	13	A binding pocket is represented by scored molecules.	Propose new molecules with better docking scores. ↑
Earth, Climate & Space	13	A weather state is estimated from a small ensemble and partial observations.	Reduce state estimation error. ↓
Statistics & Inference	13	A time series has one intervention date and no control series.	Reduce error in the treatment-effect estimate. ↓
Networks & Information Th.	12	A network link has current flows and a traffic intent.	Increase the fraction of the operator's stated requirements that are met. ↑
Software Eng. & Formal Methods	12	A test suite is generated for a program whose failures are hidden.	Increase the number of distinct failures it triggers. ↑
Signal, Imaging & Control	11	An undersea vehicle is tracked until its acoustic link drops.	Reduce position error during the outage. ↓
Computer Systems	10	A deadline job runs on spot instances that may disappear.	Reduce its worst-case competitive ratio. ↓
Numerical & Sci. Computing	10	A Hamiltonian trajectory is learned from sampled flow pairs.	Reduce long-horizon prediction error. ↓
Security & Cryptography	10	A recording and a speaker-recognition model are given.	Maximise the audio quality of a perturbation that still forces a misclassification. ↑
Neuroscience	9	Neuron positions and six target network properties are given.	Reduce the worst mismatch across those properties. ↓
Optimisation & Discrete Algorithms	9	A mixed-integer program has a fixed cutting-plane budget.	Increase the fraction of the root LP gap that is closed. ↑
Chemistry & Materials	9	A dark-field image shows a twisted bilayer crystal.	Reduce error in its atomic reconstruction. ↓
Graphics & Rendering	8	An indoor scene has a radiance cache from its original lighting.	Increase image PSNR after the lights change. ↑
Quantitative Finance	7	A Bermudan option is priced under uncertain parameters.	Reduce policy shortfall against a dual upper bound. ↓
Computational Physics	7	Magnetic diagnostics describe one tokamak discharge slice.	Reduce error in its last closed flux surface. ↓
Mechanical & Thermal	7	An immersed boundary crosses a regular grid.	Reduce field error away from the interface. ↓
Machine Learning	6	A retrieval system shares one backbone across staged checkpoints.	Increase nDCG@10 under a serving-cost budget. ↑

tion for every rating. The analyses use all 90 reviews; the supplementary materials include only the reviews whose corresponding environments are provided, as described in the reproducibility statement.

Review materials. Both groups use the same rubric and have network access, but their review materials differ. Experts receive the source paper only after submitting ratings for the eight dimensions that concern the environment itself; faithfulness to the source paper is rated last. This ordering is intended to reduce the influence of the source paper on assessments of the executable task.

Expert packets also include available solver-result summaries, which are withheld from agent judges. These summaries report scores reached by the solver models over successive sessions. Neither group receives the construction-stage verdicts or the reviewer agent's notes. The resulting expert–agent comparisons reflect differences in available evidence as well as differences in judgment.

Harness sensitivity. We also run Gemini 3.7 Flash under a second harness. Across 1,620 paired dimension ratings on 180 environments, the mean absolute difference is 0.13 points; 99% of pairs differ by at most one point. The mean ratings are 4.88 under Claude Code and 4.82 under Antigravity. This comparison indicates limited sensitivity for this judge under these two harnesses.

Overall ratings. On the expert-reviewed subset, the overall expert mean is 4.24 out of 5. Experts rate room above the starting solution (4.81) and score discrimination (4.74) highest, followed by a single objective with a stated direction of improvement (4.61). Table 9 compares expert and agent-panel means on the same environments.

We also report the proportion of environments on which the expert rating and the agent-panel mean differ by at most one point. This measure of rating proximity complements the correlations below; high values can occur when scores cluster near the top of the scale. The proportion is 96% for room above the starting solution, compared with 54% for resistance to saturation and 74% for faithfulness to the source paper.

C.2.1 Expert–Agent Disagreements

The largest mean rating gaps concern resistance to saturation ($+0.89$), shortcut resistance ($+0.58$), and faithfulness to the source paper ($+0.50$), with positive values indicating higher agent-panel ratings. Written justifications illustrate the concerns behind these gaps. The following excerpts are from human expert reviews, shortened with ellipses where indicated.

Resistance to saturation. One expert assigned the lowest rating to an environment for which the agent panel gave a high rating:

> One supplied search approach takes most of the declared improvement range. … This reaches 2.537 from 0.1504 on visible cases, about 69% of the band to 3.634, in roughly five CPU-minutes. … These results support early saturation of the declared band, without establishing that further raw accuracy gains are impossible.

The expert also cited solver results showing that models reached the target score in the first session. Agent judges did not receive those results. Reaching a target used for normalization does not establish that no further improvement is possible, so the disagreement concerns both the interpretation of the rating criterion and the evidence available to each rater. It does not resolve whether the underlying research problem is exhausted.

Shortcut resistance. Another expert described a possible route for adapting answers to grading feedback:

> This task is somewhat gameable since the model, in theory, could keep providing incorrect queries for all 50 strings and changing one by one the selection of the SQL queries … and running through the `tests/score.py` results between each iteration. The task mentions that this is not possible, but if the model can pick between the correct choices per step, it can be gameable.

This proposed attack requires access to repeated held-out grading feedback. The intended evaluation protocol withholds that feedback and removes the test files from disk before submitted code runs

(Section 3.3). Whether the route is available therefore depends on the runtime access controls, which cannot be established from the task wording alone. Separately, another review identified supplied parameters and held-out family definitions that would let a solution branch on those definitions instead of estimating the intended quantity. We treat that exposure as an environment defect. The access failures observed in solver runs are reported separately in Section 5.2 and Appendix D.4.6.

Faithfulness to the source paper. An expert assigned a rating of 2 where the agent panel assigned 4.5, noting the following simplifications:

> The task preserves 50-way zero-shot classification and the comparison between audio and text semantics. It replaces the paper's real THINGS-EEG recordings, pretrained feature pipelines, and trimodal BraVL VAE with synthetic low-dimensional arrays and a linear-Gaussian brain model. . . .

This excerpt illustrates a limitation of adapting research problems to a fixed compute budget: an environment may preserve the broad question while omitting substantive parts of the paper's method.

Judge composition. Gemini 3.7 Flash gives higher mean ratings than the other two agent judges on every dimension. Removing it lowers the overall panel mean from 4.54 to 4.39, compared with the expert mean of 4.24. Resistance to saturation and shortcut resistance remain the two largest positive gaps. Table 8 reports all dimensions under both panel compositions.

Table 8: Mean ratings on the same 90 expert-reviewed environments, using all three agent judges or excluding Gemini 3.7 Flash. Gap is the agent-panel mean minus the human-expert mean. The two largest positive gaps remain on resistance to saturation and shortcut resistance.

Dimension	Expert	3 judges	Gap	2 judges	Gap
Resistance to saturation	3.09	3.98	+0.89	3.78	+0.69
Shortcut resistance	4.09	4.67	+0.58	4.51	+0.42
Faithfulness to source paper	3.72	4.22	+0.50	3.93	+0.21
Development–test distinction	4.33	4.67	+0.34	4.54	+0.21
Research problem	4.39	4.59	+0.20	4.47	+0.08
Metric validity	4.40	4.49	+0.09	4.32	−0.08
Room above starting solution	4.81	4.89	+0.08	4.86	+0.05
Single objective	4.61	4.66	+0.05	4.61	+0.00
Score discrimination	4.74	4.65	−0.09	4.52	−0.22
All nine dimensions	4.24	4.54	+0.30	4.39	+0.15

Inter-rater correlations. Rank correlations are generally higher between agent judges than between an agent judge and the human experts (Fig. 4), although some dimensions reverse this pattern. Agent pairs are evaluated over 180 environments, while pairs involving experts use the 90 expert-reviewed environments. The differing populations and review materials limit interpretation of these comparisons. Rating agreement alone does not establish which judgments are correct.

Table 9: Human-expert and agent-panel ratings on the same 90 environments. Scores are means on a 1–5 scale; SD denotes standard deviation. Gap is the panel mean minus the expert mean, and Within 1 is the percentage of environments on which the expert rating and panel mean differ by at most one point.

Dimension	Experts	SD	Panel	SD	Gap	Within 1
Research problem	4.39	0.84	4.59	0.32	+0.20	84%
Room above starting solution	4.81	0.51	4.89	0.31	+0.08	96%
Resistance to saturation	3.09	0.96	3.98	0.84	+0.89	54%
Faithfulness to source paper	3.72	0.87	4.22	0.51	+0.50	74%
Metric validity	4.40	0.85	4.49	0.45	+0.09	87%
Single objective	4.61	0.90	4.66	0.46	+0.05	82%
Shortcut resistance	4.09	0.97	4.67	0.55	+0.58	76%
Score discrimination	4.74	0.62	4.65	0.55	−0.09	88%
Development–test distinction	4.33	0.93	4.67	0.49	+0.34	81%
All nine dimensions	4.24	0.99	4.54	0.58	+0.30	

Association with solver improvement. Among environments with both quality ratings and solver results, correlations between dimension ratings and median improvement within sessions range from -0.11 to +0.12. These small correlations do not establish a strong linear relationship in this sample. Quality ratings are not used to select or weight environments in the solver analyses.

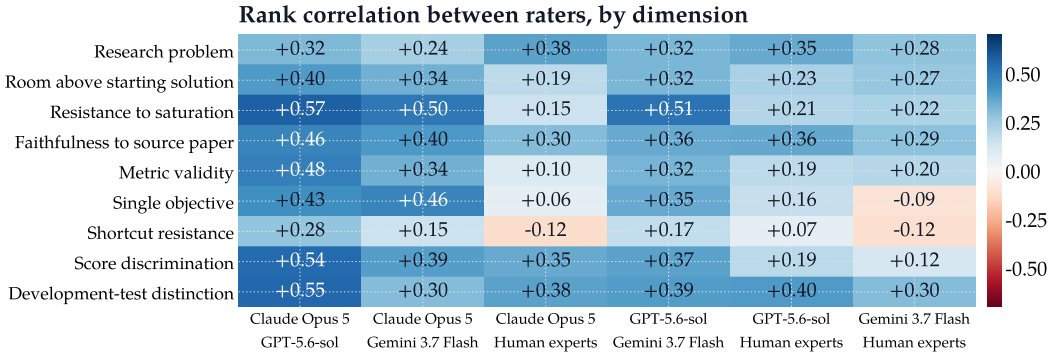


Figure 4: Rank correlations between rater pairs for each rubric dimension. Agent–agent correlations use all 180 main-run environments; correlations involving human experts use the 90 expert-reviewed environments. Many ratings are 4 or 5, producing ties and limited variation. The coefficients describe similarity in environment rankings and should be interpreted alongside the score differences in Table 9.

Table 10: Ratings from the same three agent judges on 34 manually constructed RSI-Exam tasks and the 90 RSI-Forge environments in Table 9. The RSI-Forge column repeats the agent-panel means from that table. Gap is RSI-Forge minus RSI-Exam; the 8 dimensions exclude faithfulness because this comparison has no common source-paper reference. The task sets are unpaired, so the differences do not isolate the effect of automated construction.

Dimension	RSI-Exam	RSI-Forge	Gap
Score discrimination	2.64	4.65	+2.01
Development–test distinction	3.23	4.67	+1.44
Shortcut resistance	3.92	4.67	+0.75
Single objective	4.64	4.66	+0.03
Room above starting solution	4.87	4.89	+0.02
Research problem	4.61	4.59	−0.02
Metric validity	4.57	4.49	−0.08
Resistance to saturation	4.41	3.98	−0.43
all 8	4.11	4.58	+0.46

C.2.2 Review Rubric

Human experts and agent judges use the same quality-review rubric, separate from construction acceptance checks and submission-validity checks. Table 11 reproduces the descriptions for scores 1, 3, and 5. Terms such as "starter," "ceiling," and "split" are retained from the rubric wording; in the surrounding text we use starting solution, target score, and development/test sets.

C.3 Example Environments

Table 12 gives the task definition and score references for the weather-estimation example in Section 4.3. The software-testing example below illustrates a different objective and development–test distinction.

Example selection. The software-testing environment was selected using a composite score based on the reproduced baseline's normalized score, the starting-to-target interval relative to measured noise, solver coverage, and separation of final model scores. It had the highest composite score within its field. This is a selected illustration, not a random example or an estimate of typical environment quality.

Table 11: Descriptions of scores 1, 3, and 5 in the environment-quality rubric, reproduced verbatim. The dimensions are grouped here as in the rating summaries; the human review form presents faithfulness to the source paper last.

Axis	A score of 1	A score of 3	A score of 5
Real problem	a toy with no research content	a real problem, narrowly posed	you would be pleased to see a student work on this
Headroom	the starter is already near the ceiling	a competent effort gains something	clearly large, and you can name where it comes from
Not saturable	one change takes most of the range	the first change is worth it, more remains	improvement requires several independent ideas
Faithful to the paper	unrelated to what the paper does	related, but the hard part has been removed	this is the paper's problem, honestly posed
Metric validity	measures something else	a reasonable proxy	this is the quantity the field would report
Single direction	two objectives silently traded	one quantity, the constraint is loose	one quantity, one direction, the constraint binds
Resistant to shortcuts	you can name a specific trick and say why it scores	you suspect one exists but cannot name it	you looked and the scorer closes the routes you tried
Discriminating	nothing states it and the metric looks noisy	stated, or inferable, but not justified	stated with the room to improve, and the room is the larger of the two
Split is a real axis	a uniform random partition dressed up as an axis	an axis, mild shift	a real axis; doing well on it means something

Table 12: Weather-estimation environment. The reproduced baseline is an independent implementation of the source method, evaluated under the task's interface and compute budget. Scores use the same held-out test set and normalization as in Eq. (1).

Field	Earth, Climate & Space
Task	Estimate a weather state from a small ensemble and noisy observations of half its components.
Starting solution	Ensemble Kalman analysis using the raw sample covariance, without localization or inflation.
Objective	Minimize mean state-estimation error.
Compute budget	3 h wall-clock time, 4 CPUs, and 8 GB memory for running a candidate method.
Development set	160 instances available for repeated evaluation.
Held-out test set	324 instances under forcing conditions absent from development data. Test data are loaded into evaluator memory and removed from disk before submitted code runs.
Starting-solution score	Error 1.786; normalized score 0.
Target score	Error 1.156; normalized score 1. This normalization reference need not be attainable by a solver.
Reproduced baseline	Error 1.390; normalized score approximately 0.629.

SOFTWARE ENGINEERING & FORMAL METHODS

Select 256 test configurations to discover new failure types

Source paper: Search-Based Risk Feature Discovery in Document Structure Spaces under a Constrained Budget

Objective	Maximize the fraction of reachable, previously unseen failure types discovered by a batch of configurations, averaged across campaigns.
Budget	Select at most 256 configurations after observing 100 previously evaluated configurations and their outcomes.
Held-out test set	Development campaigns use single-page documents; test campaigns use multi-page documents. Both use 20-bit configurations, but four bits change meaning to encode pagination, table continuation, and summary placement.

Task and inputs. The task models configuration selection for a pipeline that converts scanned forms into structured data. Each campaign uses a synthetic response surface representing failures under different field counts, text volumes, table sizes, noise levels, layouts, and pagination. The solver receives 100 evaluated configurations and their failure types, then selects a batch of up to 256 additional configurations. Each configuration is a 20-bit string.

Starting solution and improvement opportunities. The starting solution allocates its selections evenly across the 2^7 descriptor cells computable from the configuration. This covers known structural categories without learning which combinations trigger new failures. Improvement can involve predicting failure types from the observed examples and choosing a batch that covers distinct, previously unseen outcomes. A failure type combines the outcomes of six internal checks with three configuration-derived labels; repeated discoveries of the same type count only once.

Evaluation. For each campaign, the evaluator counts newly discovered failure types and divides by the smaller of 256 and the number of novel types reachable in that campaign's configuration space. The reachable set is computed by exhaustive evaluation of the hidden response surface, and the score is averaged across campaigns. This gives a task-specific upper reference for coverage without establishing that a solver can attain it from the supplied observations. The example measures discovery on a synthetic configuration task, rather than performance on a deployed document-processing system.

D ADDITIONAL EXPERIMENTAL DETAILS

D.1 A THREE-SESSION EXAMPLE

The aggregate scores in Table 4 summarize performance across 120 environments. Figure 5 shows one Claude Opus 5 chain in detail: every saved solution graded over three successive sessions and the changes described in the agent's records.

In this environment, a humanoid crossing stepping stones, pillars, stairs, and gaps must choose its next foothold. A submitted solution receives a local elevation map, a six-component state, and a list of admissible locations, and returns the index of one location. The evaluator measures the distance reachable from that foothold within six steps, using held-out terrain generators.

The submitted normalized score rises from 0.15 in session 1 to 0.33 in session 2 and 0.58 in session 3. The second session evaluates each foothold by several of its best continuations, replacing the first session's single-best-continuation rule. In the third session, the agent adds a check that a foothold has sufficient surrounding ground support, removing invalid continuations. With these removed, it returns to the single-best-continuation rule. The trajectory illustrates how an agent can improve by revisiting an inherited design choice.

D.2 EXPERIMENTAL SETUP

The main experiment uses 120 environments randomly sampled before score inspection from the 180 main-run environments. Four models—Claude Opus 5, GPT-5.6-sol, DeepSeek-V4-Flash, and Claude Haiku 4.5—use the same Claude Code CLI harness, with only the model endpoint changed. Each runs 3 sessions per environment, yielding 480 model–environment chains and 360 sessions per model. The original coding materials call a session a *generation*. Model weights remain fixed, and

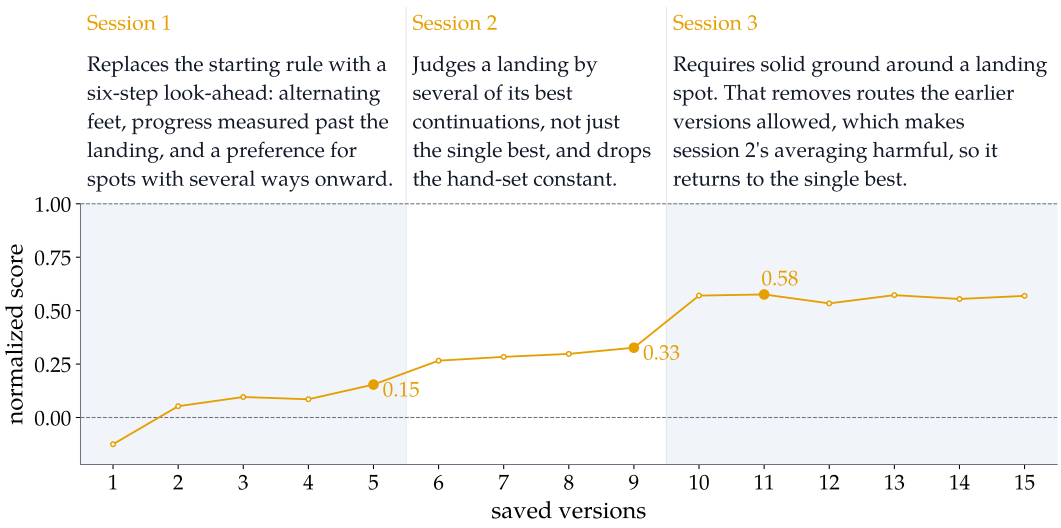


Figure 5: A three-session Claude Opus 5 chain on a foothold-selection task. Open points are saved solutions graded on the held-out test set; filled points mark final submissions, with normalized scores of 0.15, 0.33, and 0.58. The annotations summarize the changes recorded in each session. The third session adds a ground-support check and reverses the second session's averaging rule.

the models do not exchange records in this experiment. Results characterize each model under this shared harness.

Inheritance. Every session begins with a fresh context. The first receives the environment's starting solution. Later sessions receive the previous session's submitted solution, saved versions, experiment log, and handover notes. The live workspace, conversational context, and shell history are not carried over beyond these saved files. Held-out scores are not supplied to the solver.

Development and submission. During a session, the agent can edit or replace its solution and run the development evaluator without a fixed limit on evaluation calls. It saves runnable versions while working. The final submission is the solution left in the designated method directory when the session ends; this solution is inherited by the next session regardless of its score. Thus, selecting a solution is part of the task.

Stopping rule. A session ends when the agent stops or after 60 minutes, whichever comes first. The prompt states the time limit and submission path, and the harness grades the solution left at that path when the session ends. Almost all sessions end when the agent stops. This session budget is distinct from the environment's compute budget for running a candidate method, such as the limits in Table 12.

Held-out grading. Every saved version, including versions later abandoned, is replayed on the held-out test set, along with the final submission. These measurements support analyses of progress within sessions and the difference between the best saved solution and the submitted one. They are not returned to the solver during development. Access failures in the implemented environments are analyzed in Appendix D.4.6.

Table 13 lists the files used for solving, grading, and construction records. The solver-facing files are distinct from the reproduced implementation, held-out grading materials, and construction transcripts.

Starting and target scores. As defined in Eq. (1), the starting-solution score w maps to 0 and the target score u to 1. Construction records identify the evaluation data, code, or bound used to establish these values and mark provisional values as such. The target may come from a privileged-information solution, an information-theoretic or optimization bound, an exact solution on tractable instances, or a Bayes rate. It provides a reference for normalization, rather than a guarantee of attainable performance. Construction checks compare the reproduced baseline with the target; if

they are too close, the target or task conditions are revised. Solutions that perform better than the target receive normalized scores above 1.

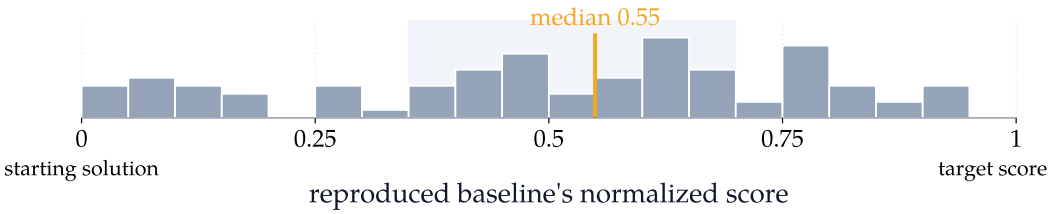


Figure 6: Distribution of reproduced-baseline normalized scores in the solver-evaluated environments. The shaded region spans the middle half of the distribution, and the line marks its median. The baseline and target are separate reference points; a target need not be attainable.

Figure 6 shows where reproduced baselines lie on the normalized scale; normalization itself does not validate a target estimate.

Reporting units. The primary outcome is the held-out score of the final submission after session 3. Scores after earlier sessions and all saved versions describe intermediate progress. A session with no valid measurement remains unscored rather than receiving an imputed score. Tables 14 and 15 summarize recorded activity for the main evaluation sample.

Table 13: Files used in environment construction and evaluation. Solver inputs, grading materials, and construction records have different access permissions. This lists the internal bundle; the supplementary release contains a subset of these materials (Appendix E.3).

Path	File	Purpose
`environment/`	`Dockerfile`	the solver image, with network access restricted to the model proxy
	`gen.py`	code for generating task data with a fixed seed
	`score.py`	automated evaluator using harness-controlled data
	`instruction.md`	the task given to a solver
	`methods/`	the starting solution and editable method directory
`solution/`	`solve.sh`	the reproduced baseline, implemented independently of the author agent
`tests/`	`hidden/`	held-out test data and grading key
	`anchors.json`	starting-solution and target scores
	`replay.py`	replay of every saved version
	`RUBRIC.md`	task-specific submission rules
	`judge.py`	model judge for submission validity
`build-traces/`		the author, reviewer and reproducer agents' transcripts

Table 14: Activity per session on the 120 main evaluation environments. Versions are saved runnable solutions that were graded; turns are assistant messages; tool calls are actions invoked by the agent. Means are reported with one standard deviation.

	Versions graded		Turns		Tool calls	
Model	Median	Mean	Median	Mean	Median	Mean
Claude Opus 5	4.0	3.6 ± 1.6	123	124 ± 30	57	59 ± 17
GPT-5.6-sol	2.0	2.7 ± 1.7	70	71 ± 17	68	70 ± 17
DeepSeek-V4-Flash	2.0	2.1 ± 1.1	126	130 ± 39	80	80 ± 24
Claude Haiku 4.5	2.0	2.7 ± 1.9	134	137 ± 40	74	75 ± 21

D.3 Improvements over Reproduced Baselines

At least one session surpasses the reproduced baseline in 68 of the 120 main evaluation environments (Section 5.2). The per-model counts are 65 for GPT-5.6-sol, 63 for Claude Opus 5, 54 for DeepSeek-V4-Flash, and 24 for Claude Haiku 4.5. These counts overlap across models, and one

Table 15: Output tokens per session on the main evaluation sample, in thousands. Means are reported with one standard deviation. Token counts describe model output volume; they are not equivalent to monetary or computational cost across providers.

Model	Median	Mean
Claude Opus 5	69.3	72.2 ± 22.8
GPT-5.6-sol	22.1	22.6 ± 8.3
DeepSeek-V4-Flash	54.5	53.8 ± 16.3
Claude Haiku 4.5	30.2	29.8 ± 8.8

environment can contribute several successful sessions. The median normalized baseline score is 0.76 in environments where no model surpasses it, compared with 0.46 where at least one does.

Example method changes. One successful solution retains alternating least squares but replaces a non-differentiable tolerance with a smooth surrogate. Another integrates over parameters that the reproduced baseline estimated at a single point.

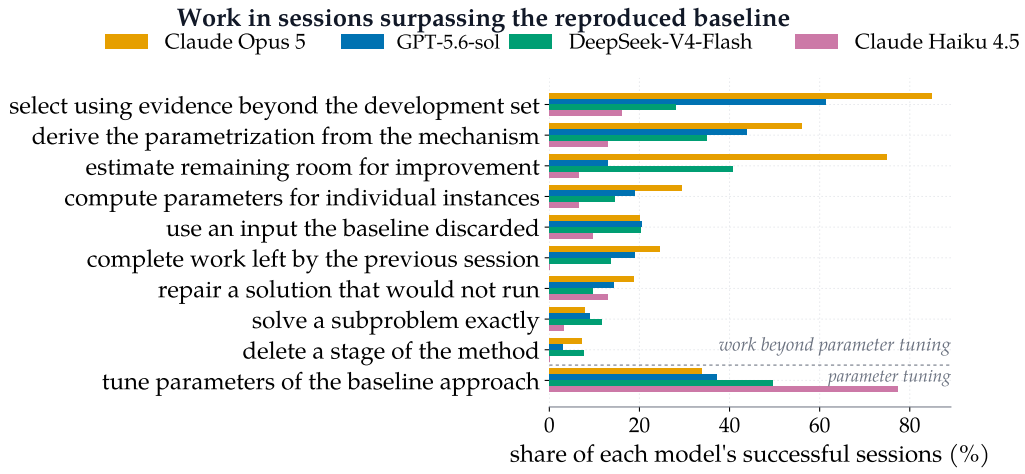


Figure 7: Transcript-coded mechanisms in sessions that surpass the reproduced baseline, expressed as percentages of each model's successful sessions. A session can receive several labels. Nine categories cover work beyond parameter tuning, and one covers tuning the baseline approach's parameters. These labels describe work within the constructed task and do not establish scientific novelty.

D.4 Behavior Coding and Analysis

We analyze solver records using three complementary coding procedures. Open coding supplies the behavior groups reported in the main text. A fixed practice codebook and additional model coders provide checks on these patterns, while a separate mechanism codebook characterizes work in sessions that surpass their reproduced baselines. These labels are model-generated and have not been validated by human annotators.

D.4.1 Coding Procedures

Open coding. Claude Opus 5 summarizes each session in a structured note covering solution changes, supporting evidence, score-directed work, unexpected events, and the stopping decision. It groups recurring practices in bounded batches without a predefined category list, retaining document identifiers through merges. Positive assignments require matching quotations from the session transcript. The resulting families are defined in Table 17.



Figure 8: Behavior families identified through open coding, grouped as improvement (21 families), exploration (27), and shortcuts (14). Each cell is the percentage of a model's 360 sessions assigned that family. Sessions can receive multiple labels. Shading is scaled separately within each group; printed percentages support comparisons across groups. Family definitions are given in Table 17.

Fixed-codebook analysis. Claude Opus 5 first receives 24 documents selected with a fixed seed and proposes practice categories without a supplied list. We then freeze the codebook and apply it to all available session records. Each positive assignment must include a matching quotation; assignments without one are discarded. The ten categories are listed in Appendix D.4.3.

Requiring a quotation makes assignments traceable, but does not establish that the interpretation is correct or that the narrated action occurred as described. Within each model, assignment counts explain little variation in output-token counts. This check does not rule out differences in narration style, so we also compare independent model codings and numeric evaluation logs below.

D.4.2 Codebooks

Table 16 separates the practice codebook, applied to session records, from the mechanism codebook, applied only to sessions that surpass their reproduced baselines. We use descriptive category labels, with the practice criteria given below.

D.4.3 Practice Codebook Definitions

The fixed practice codebook below is distinct from the open-coded families used for the behavior-group percentages in Section 5.3. The labels and definitions are paraphrased from the original coding scheme for clarity.

Table 16: Fixed codebooks for experimental practices and successful-session mechanisms. Definitions were derived from a sample and fixed before counting, with exclusion clauses for previously ambiguous cases. The practice definitions appear in Appendix D.4.3; mechanism frequencies appear in Fig. 7.

Group	Categories
Practice codebook: all coded sessions	
improvement	Checking inherited results before editing; Justifying changes from the task specification; Improving robustness without targeting the score
exploration	Bounding potential improvement before searching; Testing beyond the development set; Comparing gains with error bars; Testing controls or independent checks; Rejecting gains with unreliable evidence
shortcut	Selecting changes by development score alone
neither	Session disrupted by an environment failure
Mechanism codebook: sessions surpassing reproduced baselines	
Beyond tuning	Selecting with evidence beyond the development set; Estimating remaining room for improvement; Deriving the parametrization from the mechanism; Computing parameters for individual instances; Solving a subproblem exactly; Using an input the baseline discarded; Deleting a stage of the method; Completing work left by the previous session; Repairing a solution that would not run
Parameter tuning	Tuning parameters of the baseline approach

Category definitions. Ten categories were fixed before counting. A session may receive several labels or none.

Checking inherited results before editing Before making any edit, the agent reruns the inherited solution to reproduce its recorded score or independently derives its central claim.

Testing controls or independent checks The agent checks whether a measurement can detect the claimed effect using a deliberately weakened variant, a control with an incorrect data generator, or an independent computation.

Bounding potential improvement before searching Before searching, the agent computes an upper or lower performance bound and decomposes the remaining performance gap to allocate its budget.

Comparing gains with error bars The agent compares a change with an explicit error bar when deciding whether to accept or reject it. A before-and-after comparison alone is insufficient.

Testing beyond the development set The agent evaluates a change beyond the supplied development instances, using a stress test or a test of invariance.

Rejecting gains with unreliable evidence The agent discounts a measured gain because its supporting evidence is circular, depends on a parameter fitted to the development data, or changes sign across evaluations.

Justifying changes from the task specification A fact in the task documents or a property of the metric justifies the submitted change. A parameter-sweep result alone is insufficient.

Improving robustness without targeting the score The agent addresses failure modes such as degenerate inputs, timing limits, and fallback behavior. This work improves robustness without targeting the metric.

Selecting changes by development score alone The agent performs successive sweeps of individual parameters and treats any favorable change in the development score as an improvement.

Session disrupted by an environment failure An environment failure unrelated to the solution method consumes the session.

D.4.4 Worked Examples

The examples below illustrate the fixed practice codebook. For each behavior group, we select a session with the largest number of assigned practices in that group. These are deliberately selected examples. Quotations retain the terminology and formatting of the models' working records. One quotation is shown from each session.

Improvement. A second session by Claude Opus 5 is assigned the robustness label for repairing a timing bug.

- *Improving robustness without targeting the score*: "Also fixed a real bug either way: the wall-clock guard estimated the next restart's cost from the count of *successful* restarts, so a failed fit inflated the estimate and stopped the loop early."

Exploration. Another second session by Claude Opus 5 is assigned the control label for checking that the evaluator can detect deliberately invalid outputs.

- *Testing controls or independent checks*: "Harness falsified rather than just run: flipped answers → 0.0000, 'None' → 320 refused, starter → exactly 0.5000, and v8 imports standalone from an otherwise-empty directory."

Shortcut. A first session by Claude Haiku 4.5 receives the label below for choosing a version on the basis of a development-score change. This coding concerns the evidence for selection; it does not by itself imply a violation of the task rules.

- *Selecting changes by development score alone*: "That's worse (0.7590). The issue is that by prioritizing power cap, we're placing SUs with high power caps on the same sub-carriers, which concentrates interference. Let me revert to the 0.7818 approach:"

D.4.5 Accepting Gains Smaller than the Reported Standard Error

To complement the model-coded practices, we examine recorded scores and standard errors without judging the agents' narrative explanations.

For evaluations that report both a score and its standard error, we classify an accepted gain as smaller than the reported error when the selected version improves on the best previous version by less than the standard error attached to the selected score. This is a descriptive selection criterion, not a significance test for the difference between two correlated scores.

Evaluator formats vary, so both quantities can be recovered for only 139 sessions. The reported rates use accepted improvements in those sessions as the denominator; they do not cover every session or every saved version.

Claude Haiku 4.5 has the highest rate, at 73%, while Claude Opus 5 has the lowest, at 44%. This agrees with their ordering under the fixed-codebook analysis, although the rates differ because the two procedures measure different behaviors. The numeric criterion identifies gains smaller than a reported standard error; the codebook identifies selection based on development-score changes without sufficient supporting evidence.

D.4.6 Behavior Patterns and Sensitivity Checks

Figure 3 aggregates the families in Fig. 8 into improvement, exploration, and shortcuts. A session receives a group label when it is assigned at least one family in that group. Improvement appears frequently for all four models; exploration is more common for Claude Opus 5 than for Claude Haiku 4.5, while shortcuts show the reverse pattern. Labels overlap, so the three shares do not sum to 100%.

The groups capture differences in how agents assess changes as well as how they modify solutions. Compared with Claude Haiku 4.5, Claude Opus 5 more often uses deliberately weakened controls and tests on newly generated cases. The two middle models exchange positions across individual families (Fig. 8).

Shortcut behavior and access to grading materials

The most common shortcut families concern retaining a version because its development score increases and fitting constants to the development set. Such behavior can occur within the permitted workflow; the label reflects limited evidence that the gain will generalize.

Other families concern reading answer keys, grading code, another run's records, or reconstructing test cases from their descriptions. These labels are broader than the access-failure criterion used in Section 5.2: for example, a family may include use of labels supplied in the development set, whereas an access flag requires a record naming held-out material or another run's graded record. The coding records observed access and cannot establish that no unrecorded access occurred.

The narrower criterion flags 22 of 480 model–environment chains. We retain these chains in the main results. Excluding flagged chains gives median final normalized scores of 0.797, 0.761, 0.641, and 0.332 for Claude Opus 5, GPT-5.6-sol, DeepSeek-V4-Flash, and Claude Haiku 4.5, respectively. The ordering is unchanged and no median moves by more than 0.006. Filtering is by model–environment chain, so an access flag for one model does not exclude the other models' runs on that environment.

In 4 of the 68 environments where a model surpasses the reproduced baseline, a successful session carries an access flag. This identifies overlap between access and success without attributing the gain to that access. Behavior-group percentages remain based on the full sample; the exclusion analysis tests sensitivity of final scores.

Comparison across coding procedures

The fixed-codebook analysis measures ten predefined practices, with a supporting quotation required for every assignment. The open-coded analysis instead groups recurring behavior from structured session notes. Agreement between these analyses is informative, but neither is a human-validated reference.

The open coding yields 62 families assigned to the three behavior groups and one additional family describing submission of the inherited solution without changes. When families are ranked by their prevalence difference between Claude Opus 5 and Claude Haiku 4.5, exploration occupies ten of the eleven positions most associated with Claude Opus 5, while two shortcut families are most associated with Claude Haiku 4.5.

Additional model coders. Claude Sonnet 5, Gemini 3.7 Flash, and GPT-5.6-sol apply the fixed codebook to a stratified sample of 30 sessions per solver model. Agreement with the original coder is $\kappa = 0.73$ over the 9 categories used by either coder.

On these documents, the original coder assigns exploration to 97% of Claude Opus 5 sessions and 30% of Claude Haiku 4.5 sessions; the additional coders assign 93% and 13%, respectively. Shortcut rates show the reverse contrast: 3% versus 80% for the original coder, and 0% versus 63% for the additional coders. This reproduces the contrast on the sampled records without establishing the correctness of individual labels. These percentages use the fixed codebook and sampled subset, whereas Fig. 3 uses open-coded families across the full sample.

Work performed in successful sessions

A separate mechanism codebook labels work in sessions that surpass their reproduced baselines (Fig. 7). Across these successful sessions, 95% receive at least one of the 9 labels for work beyond parameter tuning, 5% receive only the tuning label, and 0% receive neither. These categories include activities such as repairing a non-running solution and evaluating alternatives; they do not all imply a new method.

Claude Opus 5 has the highest rate on six of the nine labels beyond tuning. Parameter tuning appears in 34% of its successful sessions and 77% of Claude Haiku 4.5's. Tuning can coexist with other work, so these rates differ from the fraction classified as tuning-only. The analysis characterizes successful sessions within the constructed tasks and does not establish which activity caused a gain or whether the result advances the source paper's field.

Table 17: Definitions of the open-coded behavior families in Fig. 8. The abbreviations *impr*, *expl*, and *shor* denote improvement, exploration, and shortcuts. Labels and coding questions are paraphrased from the original coding materials for clarity.

Group	Family	Coding question
expl	*select using reasoning beyond scores*	Did the agent keep, reject, or choose an option based on an argument about generalization, accumulated tuning choices, the scope of a change's effects, the underlying mechanism, fewer assumptions, or circular evidence for a gain?
expl	*bound potential gains before searching*	Did the agent establish an upper bound on an approach's possible gain before or instead of implementing it, and use that bound to allocate effort?
expl	*evaluate on data excluded from fitting*	Did the agent evaluate or select a solution using data excluded from fitting, rather than the data used to tune it?
expl	*override the development-score ranking*	Did the agent choose or retain an option despite a lower or indistinguishable development score, citing expected test performance or effects the evaluator does not measure?
expl	*verify results with an independent method*	Did the agent obtain the same quantity from an independent source or computation, such as a published reference, Monte Carlo estimate, numerical solver, separate implementation, analytic result, or case with a known answer, and require agreement before relying on it?
expl	*test the evaluation procedure*	Did the agent test the evaluation procedure before relying on its results?
expl	*abandon approaches after negative results*	Did the agent measure an approach, find no potential gain, and abandon it?
expl	*compare gains across individual cases*	Did the agent assess a change using paired per-instance differences, counts of wins, losses, and ties, or a requirement to improve in every subgroup, rather than two aggregate scores?
expl	*isolate contributions with ablations*	Did the agent isolate a component's contribution by removing or weakening it, or by testing separately the components of a combined change?
expl	*estimate potential gains before implementing*	Did the agent measure or bound an approach's potential gain before implementing it?
expl	*select using a validation subset*	Did the agent fit on part of the available data and use performance on the withheld part to select a solution?
expl	*select components using intermediate metrics*	Did the agent accept, reject, or choose a component or input using an intermediate statistic before or instead of evaluating a full solution?
expl	*guide changes with diagnostic measurements*	Did a diagnostic experiment, input statistic, failure breakdown, or measurement of an effect's direction guide the agent's next change, rather than reasoning alone?
expl	*perturb one factor at a time*	Did the agent compare a variant with a control differing in exactly one deliberate respect and use the resulting difference to assess that factor?
expl	*run additional generalization tests*	Did the agent construct and run an additional test of generalization beyond the development data before accepting a setting?
expl	*reject gains that do not reflect improvement*	Did the agent identify a change that would increase the score, then reject or reverse it because the gain did not reflect the intended improvement, rather than because it scored worse?
expl	*check measured cost against the budget*	Did the agent measure the cost of a run or added component and compare it with the task's resource budget?
expl	*verify checks can detect failures*	Did the agent establish that a passing check or unchanged measurement reflected what the check was intended to test?
expl	*compare gains with measured variability*	Did the agent compare a candidate's gain or fitted quantity with its measured variability, rather than a single mean or zero?
expl	*retain multiple alternatives*	Did the agent retain alternatives that it could have selected between or discarded?

continued on the next page

Group	Family	Coding question
expl	*confirm gains with another evaluation*	Did the agent require a gain to hold under another evaluation beyond the data used for tuning?
expl	*revise conclusions after contrary evidence*	Did the agent withdraw an earlier conclusion after a later measurement contradicted it?
expl	*screen on fewer cases, confirm on more*	Did the agent screen candidates on a small subset for speed and then reevaluate the remaining candidates on more instances?
expl	*investigate implausible measurements*	Did the agent defer acting on a measurement that seemed impossible, implausibly good, or inconsistent with another estimate, and investigate its cause first?
expl	*diagnose from parameter-sweep patterns*	Did the agent use the pattern of results across a fit or parameter sweep to diagnose the setup, rather than simply choosing the best-scoring value?
expl	*trace anomalies to the evaluation setup*	Did the agent investigate an implausible result or remaining error as a possible defect in the harness, inputs, assumptions, or existing code, rather than attributing it to the method or problem?
expl	*identify limits of development evaluation*	Did the agent demonstrate that the development set could not resolve a question and use other evidence to decide?
impr	*guide changes with mathematical derivations*	Did the agent derive a property mathematically and use it to choose a change, rather than relying on a measurement?
impr	*measure a baseline directly*	Did the agent run a baseline during the session to establish a comparison score, rather than reading a previously recorded score?
impr	*derive quantities instead of fitting*	Did the agent derive a quantity or property from the method's structure and use it to set, replace, or remove a component, rather than fitting it or conducting a parameter sweep?
impr	*detect a contaminated workspace*	Did the agent identify a supplied path, file, or baseline as belonging to another run or task and stop or change its approach in response?
impr	*diagnose problems through direct inspection*	Did the agent inspect concrete task artifacts and use their contents to diagnose a problem or choose a mechanism, rather than relying on score changes?
impr	*inspect environment code to understand the task*	Did the agent read environment code or construction materials, rather than the task description, to determine the objective, understand the data, or obtain code and instances?
impr	*use error analysis to guide changes*	Did the agent decompose the error and use that analysis to choose the next change or set a parameter?
impr	*verify claims against supporting files*	Did the agent treat a reported score, constant, passing check, description, or inherited assertion as unverified, inspect its supporting files, and act on what it found?
impr	*test inherited claims*	Did the agent test an inherited claim and use the result to redirect its work?
impr	*retain and re-evaluate saved solutions*	Did the agent save successive solution versions as files and use them as the experiment record?
impr	*document results for the next session*	Did the agent leave a usable record for the next session?
impr	*select configurations by constraint margins*	Did the agent select a configuration based on its measured distance from a hard constraint, rather than its score?
impr	*prepare a runnable submission*	Did the agent adapt the submission entry point so the receiving system could execute or read it without additional intervention?
impr	*inspect program behavior before editing*	Did the agent inspect or instrument the program and its outputs, then use those observations rather than a score to choose a change?
impr	*reuse information from earlier records*	Did the agent retrieve code, results, or information about earlier attempts from records left by previous sessions or other runs?
impr	*select candidates by output properties*	Did the agent accept or reject a candidate based on a property of its outputs or internal behavior, rather than its score?

continued on the next page

Group	Family	Coding question
impr	*review the previous session's record*	Did the agent search earlier session records to recover which approaches had been tried and their scores, and use that information to guide its next step?
impr	*reformulate the solution mathematically*	Did the agent change the mathematical formulation of its method while preserving the objective?
impr	*adapt parameters to individual instances*	Did the agent replace a shared setting, parameter, threshold, model, or uniform intervention with one derived from or applied to individual instances?
impr	*measure structural properties of task data*	Did the agent directly measure a property other than task performance, such as matrix commutativity, unobserved voxels, or recurring substrates across development and test data, and use it to guide its approach?
impr	*test the final submission*	Did the agent test the solution in the form in which it would be graded?
none	*submit the inherited solution unchanged*	Did the agent submit inherited code unchanged because the session's candidates failed to outperform it?
shor	*accept gains within measured variability*	Did the agent accept a configuration whose gain it identified as smaller than the variation across runs?
shor	*read the answer key*	Did the agent load answer keys or grading values present on disk and use them to diagnose or develop its method?
shor	*bound performance using hidden information*	Did the agent use information unavailable during legitimate evaluation, such as true targets, held-out information, or the best choice for each instance, to compute an upper bound for diagnosis?
shor	*infer hidden properties from task structure*	Did the agent infer an unobservable property from the stated data-generating process, disclosed task information, or harness structure and act on that inference?
shor	*design the method around evaluator logic*	Was the method shaped by the evaluator's own formulation?
shor	*read another run's workspace*	Did the agent read material belonging to other runs of the harness?
shor	*reconstruct a proxy for the test set*	Did the agent reconstruct a proxy for the held-out test set from its disclosed description and tune against it?
shor	*read packaged task files*	Did the agent leave its workspace to read the task's packaged source code or other files?
shor	*fit constants to the development set*	Did the agent repeatedly evaluate hyperparameters, thresholds, blend weights, ranges, or seeds on all or part of the development set and retain the highest-scoring choice?
shor	*assume test conditions without measurement*	Did the agent justify a constant, trigger, or component using the instructions' description of test-set composition without supporting measurements?
shor	*act on unverified claims*	Did the agent choose or rule out an approach based on a written assertion it had not verified, then plan or revise code around it without testing the assertion?
shor	*submit despite acknowledged overfitting*	Did the agent acknowledge that its constants were fitted to the development distribution and might not generalize, yet submit them unchanged without validation on held-out data?
shor	*hard-code values from development examples*	Did the submitted method contain constants, category or mechanism labels, trigger thresholds, or special cases taken from inspected development examples, rather than computing them from each input instance?
shor	*keep changes when development scores improve*	Did the agent retain or revert a change solely because the development score rose or fell, without a mechanistic explanation for the change?

D.5 Environments by Backend Model

The main construction run uses Claude Opus 5 and delivers 180 environments. To examine a second backend, we run the same sourcing queue, acceptance checks, and repair procedure with GPT-5.6-sol, producing 30 additional environments. Together these form the 210-environment corpus. We compare agent-judge ratings and solver outcomes across the two construction sets.

The comparison uses accepted environments only and does not compare construction yield. The sets are unpaired by source paper and field, so differences can reflect task composition as well as the backend model.

Quality ratings. Each of the three agent judges assigns a slightly lower mean rubric score to the second-backend set (Table 18). All differences are below 0.2 points on the five-point scale. The smaller second set and unpaired tasks limit conclusions about relative construction quality.

Table 18: Agent-judge ratings by construction backend. Cells average over rubric dimensions and then environments; n is the number of rated environments. Columns identify the backend used by the construction agents. For this comparison, Gemini 3.7 Flash uses Claude Code; Appendix C.2 compares its ratings under Claude Code and Antigravity.

	Claude Opus 5		GPT-5.6-sol	
Agent judge	Mean	n	Mean	n
Claude Opus 5 (Claude Code)	4.37	180	4.25	30
GPT-5.6-sol (Codex)	4.40	180	4.27	30
Gemini 3.7 Flash (Claude Code)	4.88	180	4.74	30

Solver performance. Table 19 compares median final submitted normalized scores on the 120 main evaluation environments and the 30 second-backend environments. All four models retain the same ordering. Score levels differ: DeepSeek-V4-Flash scores lower on the second set and Claude Haiku 4.5 higher. This consistency of ordering extends the main observation to a second construction set without isolating the backend's causal effect.

Table 19: Median final submitted normalized score after three sessions, by construction backend. The main-backend column repeats session 3 of Table 4. Here, n counts model–environment chains with all three sessions scored. The construction sets are unpaired.

	Claude Opus 5		GPT-5.6-sol	
Solver model	Median	n	Median	n
Claude Opus 5	0.800	120	0.784	30
GPT-5.6-sol	0.761	120	0.736	30
DeepSeek-V4-Flash	0.647	120	0.564	30
Claude Haiku 4.5	0.332	120	0.363	30

E Reporting and Release

E.1 Denominators

Field composition and environment-level repair rates use the 180 accepted main-run environments; expert–agent comparisons use the 90 expert-reviewed subset. The 30 second-backend environments are analyzed separately in Appendix D.5. Behavior-group percentages use the 360 sessions per model, while mechanism frequencies use only sessions surpassing the reproduced baseline. Groups and mechanisms can overlap. Access flags are counted per model–environment chain. Appendix D.4.6 gives the sample size for checks with additional model coders.

E.2 Instance Provenance

We scanned executable files in the 180 main-run environments, including `tests/`, for dataset-loading and network-access calls. Patterns covered common dataset loaders, HTTP and socket calls, and command-line downloads. The scan found 0 files with dataset-loading calls and 0 with network calls used during data generation or grading. Other detected network calls install packages during container construction or invoke the model judge for submission verification. Solver containers are configured to allow network access only to the model endpoint.

Matching operates on file text, so comments and strings can produce matches, while calls outside the patterns can be missed. Validation detected both `load_dataset` and `requests.get` in an

environment containing them. This does not establish complete coverage or rule out external data already packaged with an environment.

E.3 Release Policy

The supplementary materials provide 50 environments for inspection, including ratings and written justifications from independent human domain experts for 25 of them. We will also publicly release these environments. The full corpus, solver trajectories, and construction pipeline are not included in the supplement. The pipeline implementation will not be publicly released.

Some data files and container-build scripts are omitted to meet the submission size limit. The supplement's `DATA-OMITTED.txt` lists their paths, byte sizes, and SHA-256 hashes. The affected environments require the omitted files to be restored for execution or grading. Aggregate expert ratings use all 90 reviewed environments and cannot be recomputed from the 25 released reviews alone.

The package includes held-out evaluation materials. Exposure to these materials can compromise subsequent evaluations on the corresponding tasks. The included environments are identified in `ENVIRONMENTS.txt` so future evaluations can account for such exposure. If an environment's held-out data or grading results are used for training, that environment should be excluded from claims of performance on unseen tasks.

For version tracking, the corpus uses a SHA-256 hash computed over sorted per-environment content identifiers. This identifies the environment contents used in a run even when files are transferred without a repository commit.