LEARNING TO SELF-TRAIN: RECURSIVE SELF-IMPROVEMENT FOR CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Turning execution failures into training data requires diagnosing errors and constructing useful practice tasks. We introduce AI4AI, a teacher-assisted framework for recursive self-training. A single student model serves in three prompt-conditioned roles: a Critic diagnoses capability gaps from execution traces, a Producer constructs repository-grounded practice tasks, and a Solver generates candidate repairs. Execution-based admission checks filter the generated supervision. Joint supervised fine-tuning updates the shared model using accepted diagnosis, construction, and repair records. The updated model then helps generate supervision for the next round. Teacher demonstrations initialize all three roles; the fraction of student-generated training records increases within each role over successive rounds. On a 500-task evaluation, the final 4B model (R8) resolves 56.2% of tasks, compared with 20.2% after cold start and 55.0% for an Agent-R adaptation, the strongest external baseline in our comparison. From the cold-start model to R8, controlled evaluations show Critic diagnosis accuracy increasing from 39.0% to 78.5% on 200 held-out failure traces and Producer valid-task yield increasing from 7.0% to 31.5% over 200 proposal attempts per model. With the R8 Solver fixed, R8 Critic feedback yields a 33.5% retry success rate on 200 failed tasks, compared with 26.0% for Self-Refine-style feedback and 19.0% without feedback.

1 INTRODUCTION

The training examples a coding agent needs change as it learns. Execution traces expose errors in localization, planning, tool use, and recovery, providing a starting point for targeted practice. We study how a coding agent learns to turn this feedback into supervision for its subsequent training.

Turning a failed attempt into a training example requires diagnosing the error, constructing an executable practice task, and checking its solution. We jointly train the model to diagnose failures, construct tasks, and repair code.

AI4AI organizes recursive self-improvement into three teacher-assisted roles (Figure 1). The *Critic* reads the student's execution traces and proposes a capability gap. The *Producer* combines that gap with repository context to construct an executable task. The *Solver* attempts the task and contributes externally checked solution traces. The student uses the same model parameters for all three roles. Accepted records from these roles form $D_t = D_C \cup D_P \cup D_S$ for a joint supervised fine-tuning (SFT) update. After an update is adopted, the shared model performs all three roles in the next round and helps construct its next training set.

Teacher demonstrations supply all roles in R1–R3; later rounds introduce student data. The admission protocol specifies execution, task-validity, provenance, and overlap requirements for training records.

Self-improvement systems modify agent code (Zelikman et al., 2024; Zhang et al., 2026), learn to generate training data (Zweiger et al., 2025; Zhao et al., 2025), or couple task generation and repair (Xiao et al., 2026; Wei et al., 2026). AI4AI trains an explicit Critic alongside the Producer and Solver through joint SFT. Its diagnoses guide subsequent task construction, while student records progressively replace teacher records within each role.

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

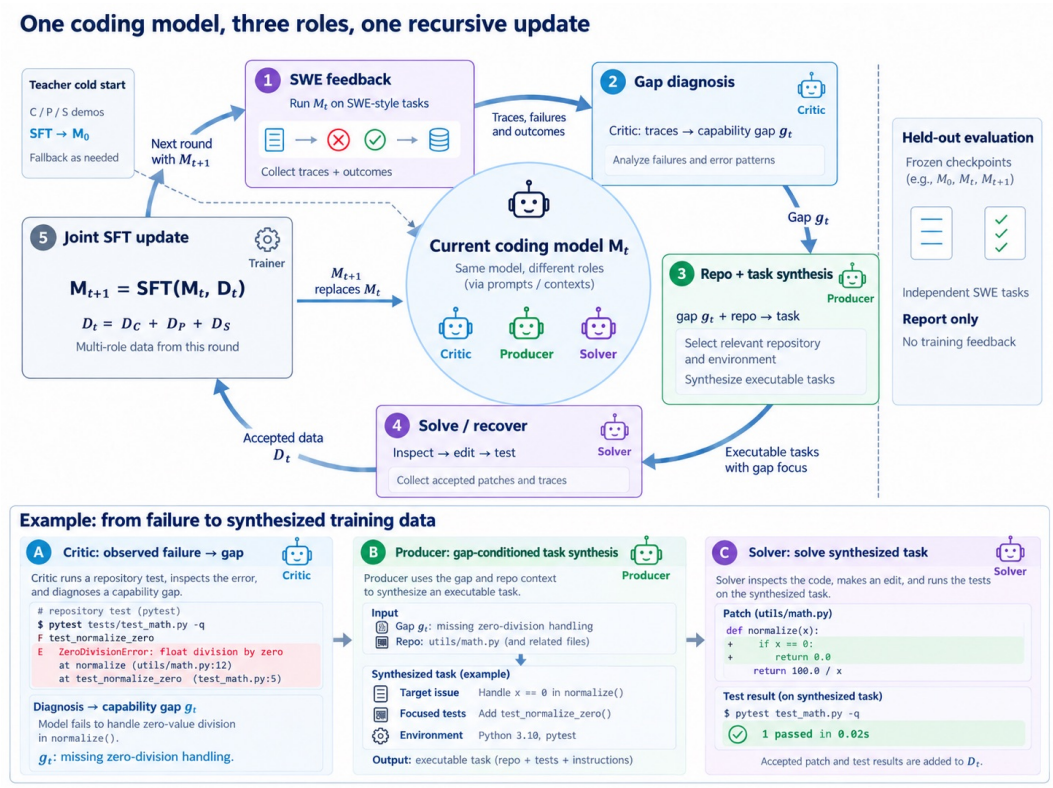


Figure 1: Learning-to-self-train loop in AI4AI: execution feedback, Critic diagnosis, Producer task construction, Solver repair/recovery, and joint SFT. An adopted model participates in the next round through the same interfaces. Teacher and student contributions are quantified in Figure 2b. The development decision determines model replacement; the optional held-out branch evaluates frozen models. Section 4 defines the evaluation sets. The lower walkthrough is a conceptual example, including its test output; Appendix F gives the specification.

On the 500-task evaluation, AI4AI reaches 59.0% at R7 and finishes at 56.2% at R8, compared with 20.2% after cold start. The final model exceeds the strongest external baseline, an Agent-R adaptation, by 1.2 percentage points (Table 1). Fixed-input evaluations show Critic diagnosis accuracy rising from 39.0% to 78.5% and Producer valid-task yield from 7.0% to 31.5%. With the final Solver fixed, R8 Critic feedback yields 33.5% retry success, compared with 26.0% for Self-Refine-style feedback.

2 Problem Setup

AI4AI alternates data construction with model updates. In each round, the agent diagnoses execution failures, constructs practice tasks, and produces solutions. Accepted teacher and student records supervise the next parameter update.

Let M_t denote the student model used at update t, with M_0 initialized by supervised fine-tuning (SFT) of Qwen3.5-4B-Base on teacher demonstrations. The teacher continues to contribute supervision after initialization. Each record identifies its role (Critic, Producer, or Solver) and the teacher or student model that generated it. Section 4 gives the model and runtime configurations.

The student roles share parameters but receive different inputs. The Critic analyzes task-visible execution traces and outcomes, without hidden repairs. The Producer receives a diagnosed gap, eligible repository context, and task constraints. The Solver receives a synthesized task and workspace, without a reference patch. Recovery is a Solver mode that continues from an actual failed prefix. Figure 1 shows the three-role design.

The protocol defines an adaptive feedback set $\mathcal{F}$ and an optional, isolated reporting set $\mathcal{H}$. Executions on $\mathcal{F}$ guide diagnosis, task allocation, and model selection. When used, $\mathcal{H}$ evaluates frozen models; its tasks, traces, labels, and scores are excluded from subsequent updates. Section 4 specifies the scope of the 500-task results.

Training requests run from the cold start R1 through R8, reported as $M_0 = \mathrm{R1}$ through $M_7 = \mathrm{R8}$. The protocol records each update's retained parent, including after a candidate is rejected.

The accounting units are failure traces, diagnoses, proposed tasks, accepted tasks, SFT records, and supervised tokens. Tasks and records have a one-to-many relationship; each record separates masked context from supervised assistant actions. The admission protocol tracks each unit.

3 Learning to Self-Train with Three Roles

Each AI4AI round starts from the current model's execution feedback. The role-source policy selects teacher or student outputs for supervision, and an SFT update trains a candidate for the next round. The procedure has five steps: feedback collection, gap diagnosis, task construction, solution generation, and model selection. Execution gates filter tasks before solution generation and training.

Each role contributes a different supervised target. Critic records pair execution traces with diagnoses supported by specific trace spans. Producer records pair diagnosed gaps and repository context with task specifications, code mutations, and focused checks. Solver records contain the actions used to repair the resulting code. These records train one shared model under role-specific prompts to diagnose failures, construct tasks, and repair code.

3.1 Bootstrapping the self-training roles

Teacher demonstrations of Critic, Producer, and Solver behavior initialize M_0 through SFT. Student records are generated by a shared checkpoint under role-specific prompts. Teacher and student outputs use the same interfaces, and each record identifies its generating actor. All role data are teacher-generated through R3. Student data first enter Critic in R4, Producer in R5, and Solver in R6 (Table 5). The equations below describe the student branch.

3.2 Step 1: collect model feedback

The student executes adaptive tasks using an agent interface and execution harness. The feedback packet comprises the task description, model actions, tool observations, visible test output, and terminal status. The diagnostic protocol excludes reference repairs and hidden grader material and requires infrastructure failures to be separated from model failures. The Critic reads the attempt together with its task and execution context.

3.3 Step 2: diagnose capability gaps

For feedback task $f_i \in \mathcal{F}$, the student Critic proposes a gap:

$$g_{ti} = \mathrm{Critic}_{M_t}\big(\mathrm{Blind}(\mathrm{Execute}(M_t, f_i))\big). \tag{1}$$

Here $\mathrm{Execute}$ returns the model's interaction trace and outcome; Blind removes information unavailable to the agent under the diagnostic protocol. The output g_{ti} contains a proposed gap and supporting trace spans, or an abstention. Task generation is conditioned on the behavior identified in the diagnosis.

Gap categories include execution control/recovery and planning/decomposition. The gap-to-budget policy maps admitted diagnoses to repository contexts and proposal allocations (Algorithm 1).

3.4 Step 3: synthesize and admit executable tasks

A generated task must expose the intended behavior through an executable check. The Producer receives an admitted gap and repository context, then proposes a task specification, focused checks,

and a controlled code mutation. Let c be the clean code state, μ the proposed mutation, and J the focused test procedure. $\mathrm{Restore}$ restores the pre-mutation code. The execution gate is defined by

$$J(c) = \text{pass}, \qquad J(\mu(c)) = \text{fail}_{\text{signature}}, \qquad J(\text{Restore}(\mu(c))) = \text{pass}. \tag{2}$$

The failure signature identifies the intended test failure. The gate requires fixed dependencies and commands, stable execution, and exact restoration of the source state. Repeating the test before mutation, after mutation, and after restoration attributes the focused failure to the code change. Production accounting includes rejected proposals.

3.5 Step 4: solve, recover, and assemble data

The Solver receives the task and defective workspace without a reference patch. It edits and tests candidate repairs; recovery continues from a failed Solver prefix. Under the admission protocol, successful trajectories must also satisfy provenance, overlap, and restoration checks before entering training. Failed prefixes provide recovery context; the loss targets admitted assistant actions that follow them.

Accepted diagnostic, construction, and solution records form D_C, D_P, and D_S, respectively:

$$D_t = D_C \cup D_P \cup D_S, \qquad M_{t+1} = \mathrm{SFT}(M_t, D_t). \tag{3}$$

Here M_{t+1} denotes the candidate before adoption. Role-record counts vary by round (Appendix D); record sampling and token exposure determine each role's contribution to training.

For record i, let x_i be its conditioning context, y_{ij} its jth token, and $m_{ij} \in \{0, 1\}$ the target mask. For a batch containing at least one supervised token, the token-normalized objective is

$$\mathcal{L}(\theta) = -\frac{\sum_{i,j} m_{ij} \log p_\theta(y_{ij} \mid x_i, y_{i,<j})}{\sum_{i,j} m_{ij}}. \tag{4}$$

System and user messages, tool observations, and failed prefixes are masked from the loss. Admitted assistant targets from all three roles update the shared parameters.

Solver targets train repair actions; Critic and Producer targets train the diagnoses and task specifications used to construct subsequent training data. After adoption, all three student roles use the updated parameters to generate the next round's data.

3.6 Step 5: select the next model

The protocol uses development tasks to decide whether to adopt a candidate. The adoption rule is an input to Algorithm 1: an accepted candidate replaces the incumbent; otherwise, the incumbent is retained. When an isolated reporting set $\mathcal{H}$ is used, it evaluates frozen checkpoints without feeding back into diagnosis, generation, training, or adoption.

4 Experimental Setup

We evaluate coding performance, the quality of Critic diagnoses and Producer tasks, and the effect of Critic feedback on repair success. For each training round, we record the amount and source of supervision.

4.1 Models and recursive training sequence

Starting from Qwen3.5-4B-Base, we evaluate eight training rounds, R1–R8. R1 is the Base-to-SFT cold start, followed by seven recursive updates. Teacher-generated data supply all three roles through R3. Student-produced Critic data enter in R4, Producer data in R5, and Solver data in R6. Table 5 reports the source mix within each role, separately from role-record counts and supervised-token volumes.

The agent runs in OpenHands (Wang et al., 2024). The teacher API identifier is `deepseek-ai/deepseek-v4-flash-0731`. Table 2 gives the model, runtime, and training configuration.

Algorithm 1 Recursive self-training update in AI4AI

Require: Student M_t; teacher; adaptive tasks $\mathcal{F}$; repositories; role prompts; source-selection and gap-allocation policies; admission criteria; resource limits; development adoption rule

Execute M_t on $\mathcal{F}$; separate infrastructure failures.
Select generating actors under the role-source policy.
Critic: inspect feedback; propose supported gaps or abstain.
Allocate practice under the specified gap-to-budget policy.
Producer: construct tasks; apply execution and admission gates.
Solver: attempt repairs/recoveries; externally check outcomes.
Admit role records separately; assemble $D_C \cup D_P \cup D_S$.
if an update-level admission or resource requirement fails **then**
    Retain M_t.
else
    Train a candidate from M_t using the admitted records.
    Evaluate on development tasks; adopt only if the rule accepts.
end if
If an isolated $\mathcal{H}$ exists, report frozen-model scores without feedback.

4.2 Evaluation sets and reporting units

The main evaluation reports resolved tasks out of 500 for each model, referred to as the *500-task evaluation*. We report resolved-task counts and resolution rates for the base model, all eight training rounds, and the comparison methods.

Comparison methods. Starting from the 4B base checkpoint, we compare AI4AI with Socratic-SWE (Xiao et al., 2026), SFT on SWE-smith data (Yang et al., 2025), Self-play SWE-RL (Wei et al., 2026), and Agent-R adapted to the coding setting (Yuan et al., 2025). Table 1 reports the final-model results from these experiments on 500 tasks.

4.3 Controlled role evaluation and feedback utility

We compare the cold-start model M_0 (R1), R4, R6, R8, and the Teacher. For Critic evaluation, every model diagnoses the same 200 failure traces, excluded from training and adaptive feedback, with one diagnosis per trace. A diagnosis is correct when it identifies a key failure cause supported by trace evidence and contains no factual error that changes the conclusion. Correct-and-actionable feedback also gives specific advice that can be carried out in the given repository and tool environment. Two human raters independently score outputs with model identities hidden; a third adjudicates disagreements. Both rates use all 200 traces as the denominator. Actionability is judged from the advice and repository context; repair success is measured separately in the retry experiment.

For Producer evaluation, each model receives the same 50 gap-description and repository-snapshot conditions and makes four proposals per condition. Prompts, tool permissions, token limits, and execution budgets are held fixed. All 200 attempts count, including malformed outputs, timeouts, and empty responses. A runnable task loads and executes its tests. Causal checks require the original version to pass, the changed version to fail, and the reverted version to pass again. A valid task must also match the target gap under blinded assessment and survive within-model deduplication. Only independently generated outputs count as Student successes. Runnable rate and valid-task yield both use 200 as the denominator. Production-log acceptance is measured separately.

For feedback utility, the R8 Solver retries the same 200 failed tasks from their failure states under four conditions: no feedback, M_0 Critic feedback, Self-Refine-style feedback, or R8 Critic feedback. In the Self-Refine-style condition (Madaan et al., 2023), the Solver generates its own feedback before retrying. Each condition allows one retry per task with identical retry token, tool-call, and time limits. Appendix C.1 provides the counts for the role and feedback evaluations.

Training-source accounting. We count each final-training record once, regardless of the number of training epochs, and assign it to one source. Teacher-generated and Teacher-rewritten records count as Teacher data; independently generated student records count as Student data. No records

Table 1: Final-model results for our implementations of the comparison methods on the 500-task evaluation.

Method	Resolved/500	Rate (%)
Base 4B	36	7.2
Socratic-SWE	225	45.0
SFT on SWE-smith	240	48.0
Self-play SWE-RL	260	52.0
Agent-R adapted	275	55.0
AI4AI (R8)	**281**	**56.2**

have mixed or unknown sources. Within each role, student contribution is the fraction of final-training records generated independently by the student.

4.4 Metrics

Resolution is the number of tasks labeled successfully resolved by the evaluation divided by its denominator N. For a pair evaluated on identical tasks, let $y_i^a, y_i^b \in \{0, 1\}$ denote candidate and reference outcomes. Then

$$\widehat{\Delta} = \frac{1}{N} \sum_{i=1}^{N} (y_i^a - y_i^b) = \frac{w - \ell}{N}, \tag{5}$$

where w and ℓ count candidate-only and reference-only successes. The percentage-point difference is $100\widehat{\Delta}$. The main results report marginal counts and aggregate percentage-point differences. Eq. equation 5 applies to paired task-level outcomes.

4.5 Process measurements

For each training round, production logs record proposals, runnable tasks, causal-check passes, and Solver-accepted tasks. Appendices D and E report training records by role and source, supervised-token counts, inference use, and stage-specific API charges.

5 Results and Analysis

5.1 Comparison with other training methods

AI4AI resolves 281 of 500 tasks (56.2%; Table 1), compared with 275 (55.0%) for Agent-R, the strongest external baseline. The difference is six tasks, or 1.2 percentage points. Relative to Socratic-SWE, SFT on SWE-smith, and Self-play SWE-RL, the gains are 11.2, 8.2, and 4.2 percentage points, respectively.

5.2 Coding performance and participation in self-training

Performance across updates. Resolution rises from 7.2% at Base to 20.2% after cold start and 56.2% at R8 (Figure 2a), a 36.0-percentage-point increase beyond cold start. From R1 to R3, when all training data come from the Teacher, resolution increases by 29.2 percentage points. R8 finishes 6.8 percentage points above R3. Performance varies across updates: R6 falls to 55.2%, R7 reaches the sequence maximum of 59.0%, and R8 finishes 2.8 percentage points below R7. The final R8 model is used for the external comparison and final-role evaluations.

Student contributions to training data. Student data enter Critic, Producer, and Solver at R4, R5, and R6, respectively, reaching 90.2%, 80.0%, and 50.0% at R8 (Figure 2b). Each fraction uses that role's final-training records as its denominator (Table 10). At R8, student-generated data supply most Critic and Producer supervision and half of Solver supervision.



Figure 2: Coding performance and student participation across training rounds. (a) Resolution on 500 tasks after each training round, with Base as reference. (b) Independently generated Student records as a fraction of each role's final-training records, beginning at R1. Exact counts appear in Tables 3 and 10.

5.3 Quality and utility of student-generated supervision

Figure 3 compares Critic diagnoses and Producer tasks on fixed evaluation inputs, then measures repair success with a fixed Solver under four feedback conditions.

Diagnoses become more accurate and actionable. Critic diagnosis accuracy rises from 78/200 (39.0%) at cold start to 157/200 (78.5%) at R8, while correct-and-actionable feedback rises from 49/200 (24.5%) to 137/200 (68.5%; Figure 3a). Both measures improve across the evaluated Student models, gaining 39.5 and 44.0 percentage points, respectively. The Teacher reference reaches 87.0% accuracy and 79.0% correct-and-actionable feedback.

The Producer generates more valid tasks. The Producer's runnable rate rises from 43/200 (21.5%) to 103/200 (51.5%), while valid-task yield rises from 14/200 (7.0%) to 63/200 (31.5%; Figure 3b), a 24.5-percentage-point gain in tasks that pass causal checks, match the target gap, and survive deduplication. Causal-pass counts rise from 25 to 84 (Table 7). Under the same generation conditions, the Teacher yields 91/200 valid tasks (45.5%).

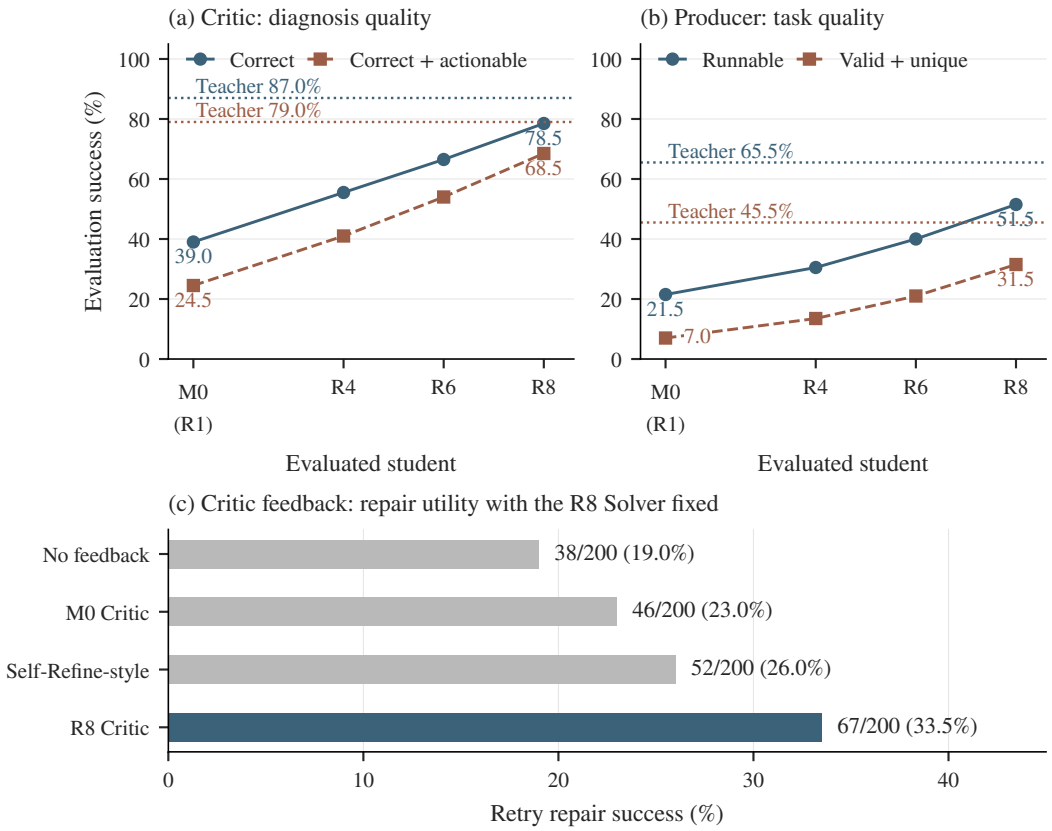


Figure 3: Quality and utility of student-generated supervision. (a) Correct diagnoses and correct-and-actionable feedback on 200 fixed failure traces. (b) Runnable and valid, unique tasks from 200 attempts over 50 fixed conditions. In (a,b), dotted lines are Teacher references; connected points are the evaluated Student models. (c) Repair success on the same 200 failed tasks under four feedback conditions, with the R8 Solver and retry budgets fixed. Exact counts appear in Tables 6–8.

Critic feedback helps a fixed Solver repair failures. Without feedback, the R8 Solver repairs 38/200 tasks (19.0%). It repairs 46/200 (23.0%) with cold-start Critic feedback, 52/200 (26.0%) with Self-Refine-style feedback, and 67/200 (33.5%) with R8 Critic feedback (Figure 3c). R8 Critic feedback yields 15 more repairs than self-feedback, a gain of 7.5 percentage points, and improves recovery by 14.5 percentage points over no feedback. All four conditions use the same failure states, Solver checkpoint, and retry limits.

5.4 Production and Supervision Accounting

Task acceptance. R6–R8 accept 22/128, 30/150, and 42/180 proposals (17.2%, 20.0%, and 23.3%; Figure 4a). The runnable fraction rises from 26.6% to 33.3%, while final acceptance among causal-pass tasks rises from 64.7% to 76.4%. The causal-check pass rate among runnable tasks falls from 100.0% to 91.7% (Figure 4b). The runnable stage rejects the largest fraction of its inputs across all eight rounds. R8 accepts 42 of 180 proposals (23.3%), compared with 82 of 980 (8.4%) at R5.

Supervision and resource accounting. Figure 5 and Tables 9–12 give the record and token counts. The R6–R8 final-training sets contain 334, 533, and 712 records, respectively, summed across the three roles. Per-task inference means and stage-specific API estimates appear in Figure 6 and Tables 13–14.

6 Related Work

Software engineering agents and environments. SWE-bench evaluates repository-level issue resolution (Jimenez et al., 2024). SWE-agent and OpenHands provide interfaces for tool-using software agents (Yang et al., 2024; Wang et al., 2024). Agentless separates localization, repair, and patch validation (Xia et al., 2025); AutoCodeRover uses structured code search to guide patch generation (Zhang et al., 2024). SWE-Gym, R2E-Gym, and SWE-smith expand the supply of executable training tasks (Pan et al., 2025; Jain et al., 2025; Yang et al., 2025).

Training data and feedback. Self-Instruct filters generated instructions, and STaR trains on rationales that yield correct answers (Wang et al., 2023; Zelikman et al., 2022). SWE-RL trains on software-evolution data with patch-similarity rewards (Wei et al., 2025). Self-Refine and Reflexion use feedback without parameter updates (Madaan et al., 2023; Shinn et al., 2023); RISE and Agent-R train models to revise unsuccessful attempts (Qu et al., 2024; Yuan et al., 2025). We compare final coding performance with an Agent-R adaptation and compare trained Critic feedback with Self-Refine-style feedback using a fixed Solver (Table 1; Figure 3c).

Recursive self-improvement. STOP recursively improves an LM-based code improver, while Darwin Gödel Machine evolves coding agents through self-modification and an archive of agent variants (Zelikman et al., 2024; Zhang et al., 2026). Both modify code around fixed language models. SEAL instead learns to generate self-edits for weight updates using post-update performance as feedback (Zweiger et al., 2025). Absolute Zero jointly trains a task proposer and solver with execution-based feedback (Zhao et al., 2025). AI4AI updates shared Critic, Producer, and Solver parameters through teacher-assisted SFT on repository interactions.

Self-evolving coding agents. Socratic-SWE derives skills from traces and trains a shared Generator/Solver policy with gradient-aligned generator rewards (Xiao et al., 2026). Self-play SWE-RL jointly trains bug injection and repair, including bugs derived from failed repairs (Wei et al., 2026). AI4AI trains an explicit Critic alongside the Producer and Solver using joint SFT. We evaluate diagnosis quality, task validity, and feedback utility across student checkpoints. Table 1 compares final coding performance with Socratic-SWE, Self-play SWE-RL, and SFT on SWE-smith data.

Data selection. DoReMi learns data-mixture weights with a proxy model; LESS selects instruction data using optimizer-aware gradient similarity (Xie et al., 2023; Xia et al., 2024). AI4AI uses diagnosed capability gaps to construct new practice tasks.

7 Discussion and Conclusion

Learning to self-train. AI4AI trains a coding agent to diagnose failures, construct practice tasks, and repair code. Accepted records from all three roles update the shared model, which then helps produce supervision for the next round. By R8, independently generated student records account for 90.2%, 80.0%, and 50.0% of Critic, Producer, and Solver training records, respectively.

Conclusion. The final AI4AI model resolves 56.2% of 500 tasks, compared with 20.2% after cold start and 55.0% for the strongest external baseline. R7 achieves the highest score in the training sequence, 59.0%. Controlled evaluations show gains in diagnosis accuracy and valid-task yield. With the R8 Solver fixed, R8 Critic feedback raises retry success by 7.5 percentage points over Self-Refine-style feedback and 14.5 percentage points over no feedback.

Reproducibility Statement

Section 3 specifies the role interfaces, execution gate, training objective, and recursive self-training procedure. Section 4 gives the configuration and evaluation scope. Appendices A and B specify the record and evaluation requirements; Appendices D and E provide record and resource accounting. Appendix C lists aggregate results, plotting code, and figure data.

Ethics Statement

Repository-derived code is subject to its source licenses and access restrictions. Generated tasks introduce defects into executable workspaces. Generated defects and repairs require authorized, isolated execution and security review before deployment. Artifact release requires review of repository and model licenses, redistribution rights, and sensitive material. The reporting protocol retains negative outcomes, rejected tasks, and superseded evaluations.

AI Use Statement

During the preparation of this article, the authors used ChatGPT for language editing. The authors reviewed and revised the resulting text and take full responsibility for the content of the publication.

References

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *arXiv preprint arXiv:2504.07164*, 2025. URL https://arxiv.org/abs/2504.07164v1.

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. URL https://arxiv.org/abs/2310.06770v3.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-Refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, volume 36, 2023. doi: 10.52202/075280-2019. URL https://doi.org/10.52202/075280-2019.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *International Conference on Machine Learning*, 2025. URL https://arxiv.org/abs/2412.21139v2.

Yuxiao Qu, Tianjun Zhang, Naman Garg, and Aviral Kumar. Recursive introspection: Teaching language model agents how to self-improve. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1754. URL https://doi.org/10.52202/079017-1754.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023. URL https://arxiv.org/abs/2303.11366v4.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024. URL https://arxiv.org/abs/2407.16741v3.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-Instruct: Aligning language models with self-generated instructions. *arXiv preprint arXiv:2212.10560*, 2023. URL https://arxiv.org/abs/2212.10560v2.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2629. URL https://doi.org/10.52202/085713-2629.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward training superintelligent software agents through self-play SWE-RL. In *International Conference on Machine Learning*, 2026. URL https://arxiv.org/abs/2512.18552v3.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Demystifying LLM-based software engineering agents. *Proceedings of the ACM on Software Engineering*, 2(FSE), 2025. doi: 10.1145/3715754. URL https://doi.org/10.1145/3715754.

Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. LESS: Selecting influential data for targeted instruction tuning. *arXiv preprint arXiv:2402.04333*, 2024. URL https://arxiv.org/abs/2402.04333v3.

Chuan Xiao, Zhengbo Jiao, Shaobo Wang, Wei Wang, Bing Zhao, Hu Wei, Linfeng Zhang, and Lin Qu. Socratic-SWE: Self-evolving coding agents via trace-derived agent skills. *arXiv preprint arXiv:2606.07412*, 2026. URL https://arxiv.org/abs/2606.07412v1.

Sang Michael Xie, Hieu Pham, Xuanyi Dong, Nan Du, Hanxiao Liu, Yifeng Lu, Percy Liang, Quoc V. Le, Tengyu Ma, and Adams Wei Yu. DoReMi: Optimizing data mixtures speeds up language model pretraining. *arXiv preprint arXiv:2305.10429*, 2023. URL https://arxiv.org/abs/2305.10429v4.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793v3.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025. URL https://arxiv.org/abs/2504.21798v2.

Siyu Yuan, Zehui Chen, Zhiheng Xi, Junjie Ye, Zhengyin Du, and Jiecao Chen. Agent-R: Training language model agents to reflect via iterative self-training. *arXiv preprint arXiv:2501.11425*, 2025. URL https://arxiv.org/abs/2501.11425v3.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. STaR: Bootstrapping reasoning with reasoning. *arXiv preprint arXiv:2203.14465*, 2022. URL https://arxiv.org/abs/2203.14465v2.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation. In *Conference on Language Modeling*, 2024. URL https://arxiv.org/abs/2310.02304v3.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2505.22954v3.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024. doi: 10.1145/3650212.3680384. URL https://doi.org/10.1145/3650212.3680384.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3534. URL https://doi.org/10.52202/085713-3534.

Adam Zweiger, Jyothish Pari, Han Guo, Yoon Kim, and Pulkit Agrawal. Self-adapting language models. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-2483. URL https://doi.org/10.52202/085713-2483.

A Record and Execution Protocol

The protocol defines record fields and acceptance checks for each stage of recursive self-training. Aggregate measurements appear in Appendices D and E.

A.1 Model, role, and actor identity

Required record fields are the iteration checkpoint, role prompt, generating actor, and source artifact. The checkpoint identifies M_t; the role is Critic, Producer, or Solver. The actor is the student checkpoint or the teacher supplying scheduled or fallback data. The source artifact links the record to its trace, repository, task, and admission receipt.

The release manifest must record exact model revisions, tokenizer, chat template, serving configuration, OpenHands version, native-tool parser, Docker image, exporter, and grader. Teacher fallback requires a terminal reason and policy code; retries and actor replacements require explicit provenance.

A.2 Feedback and diagnosis records

Each feedback packet requires the task ID, checkpoint hash, prompt and tool configuration, complete trace of visible actions and observations, terminal state, grader linkage, and infrastructure classification. Hidden repairs and held-out labels are excluded by the protocol. A Critic record requires trace citations, taxonomy version, proposed gap labels, confidence or abstention, and actor identity. Reclassifying a provider, environment, harness, or grader failure as a capability gap requires an independent adjudication record.

A.3 Task synthesis and causal admission

A Producer proposal must identify its parent gap, repository digest, clean source state, mutation, focused command, expected failure signature, restoration procedure, and task surface. Admission requires reproducing the clean-pass–mutated-fail–restored-pass sequence in a network-disabled environment. Source hashes before mutation and after restoration must match. The receipt must record every attempted candidate, including parse failures, timeouts, invalid signatures, and restoration failures.

The gate tests the mutation's effect on the focused check. Admission also requires complete provenance and compliance with licensing and repository policy. Tasks overlapping a protected partition, duplicating an admitted task, or exposing a reference repair are excluded.

A.4 Solver execution and SFT assembly

The Solver receives the admitted task and defective workspace without a reference patch. A recovery example begins from an actual failed Solver prefix and remains part of the Solver role. External grading, test execution, provenance, source restoration, and leakage checks determine acceptance.

The SFT record must preserve the visible messages and target mask. System and user messages, tool observations, and failed prefixes are context. Only the admitted assistant actions and end markers are supervised. For each iteration, the release manifest must reconcile unique tasks, trajectories, records, single-pass target tokens, repeated exposure, and consumed target tokens.

A.5 Update and checkpoint lineage

A completed training-update record must identify the parent checkpoint, release digest, optimizer, scheduler, distributed reduction, precision, and devices. It must also record loss and gradient finiteness checks, parameter changes, the target-preserving stopping rule, checkpoint shards, tokenizer artifacts, and serving verification. Configured budgets, dataset target-token counts, and consumed training tokens are recorded separately. The lineage records each candidate's actual parent, including the retained checkpoint after a rejected update.

Table 2: Model, runtime, and training configuration. Caps denote configured limits.

Configuration	Setting	Configuration	Setting
Initial model	Qwen3.5-4B-Base	Runtime	OpenHands 0.53.0
Context limit	32,768 tokens	Temperature	0
Epoch cap	2	Output cap	4,096 tokens†
Learning rate	5×10^{-6}	Iteration cap	100†
Warmup / decay	0.05 / 0.1†	Tool interface	Native tool calling
Gradient cap	1.0	Training seed / ranks	42 / 4

†Unspecified fields: output-cap scope (per call or per trajectory), iteration unit, and warmup/decay conventions.

A.6 Quarantine and supersession

Artifacts affected by parser, condenser, harness, grader, or provenance defects require quarantine identifiers. Regeneration or regrading must document the correction and link each revised status to the original task. Accounting retains superseded, quarantined, and rejected records; the canonical release and formal comparisons exclude them.

B Evaluation and Selection Protocol

B.1 Adaptive development comparisons

Each paired report must contain the same ordered task IDs for incumbent and candidate, identical inference and tool configuration, raw rollouts, predictions, official grader linkage, infrastructure status, and terminal reason. Candidate-only wins, incumbent-only wins, both-success ties, and both-failure ties sum to the common denominator. A policy fixed before comparison determines how missing and infrastructure-invalid tasks are handled; task inclusion must be independent of the observed outcomes.

Results are reported as aggregate success rates and percentage-point differences.

B.2 Held-out reporting

The held-out protocol requires a manifest fixed before evaluation and checked against training repositories, synthesized tasks, adaptive tasks, and historical use. Frozen checkpoints are evaluated with no return of task identities, traces, labels, or scores to diagnosis, generation, training, early stopping, or promotion. The exposure record determines whether an evaluation qualifies as held-out or adaptive.

C Configuration and Aggregate Measurements

Table 2 gives configured limits. Exact resolved-task counts, production-stage counts, and within-role source fractions appear in Tables 3, 4, and 5, respectively.

Locating production losses. Let P, E, C, and A denote the reported proposal, runnable, causal-pass, and Solver-accepted counts for a request. For the nonzero counts in Table 4, the accounting identity $A/P = (E/P)(C/E)(A/C)$ separates execution, causal checking, and Solver acceptance. In R6 these factors are $34/128$, $34/34$, and $22/34$; in R8 they are $60/180$, $55/60$, and $42/55$. The runnable stage discards the largest fraction of its incoming tasks in every reported request. R6–R8 improve the runnable and final Solver fractions while the conditional causal-pass fraction decreases.

Plot reproducibility. The `figures/` directory contains `plot_results.py` (Python, Matplotlib, and NumPy) and its JSON inputs. `comparison_results_data.json` supplies the corrected R8 count and four-condition feedback results, overriding earlier entries during plotting. The JSON files record aggregate counts and source descriptions; null denotes missing values. The core schematic is supplied separately.

Table 3: Resolution across training requests on the 500-task evaluation. R1 is the cold start; R2–R8 are successive updates. Change is the percentage-point difference from the preceding row.

Model	Request	Resolved/500	Rate (%)	Change (pp)
Original base	—	36	7.2	—
Cold start M_0	R1	101	20.2	+13.0
M_1	R2	197	39.4	+19.2
M_2	R3	247	49.4	+10.0
M_3	R4	268	53.6	+4.2
M_4	R5	292	58.4	+4.8
M_5	R6	276	55.2	−3.2
M_6	R7	**295**	**59.0**	+3.8
M_7	R8	281	56.2	−2.8

Table 4: Production counts. Feedback and Diagnoses count failure traces and valid diagnoses, respectively. Historical release-task counts use a separate reporting scope (Appendix D).

Request	Feedback	Diagnoses	Proposals	Runnable	Causal pass	Solver pass
R1	183	120	980	120	95	72
R2	216	145	1,020	145	102	81
R3	216	158	920	158	96	70
R4	216	171	1,050	171	110	85
R5	216	180	980	180	108	82
R6	98	98	128	34	34	22
R7	120	108	150	45	42	30
R8	150	132	180	60	55	42

Table 5: Teacher-generated or Teacher-rewritten fraction of final-training records within each role, rounded to one decimal. Student is the complement because mixed and unknown sources are zero. Percentages use the record counts in Table 10.

Request	Critic (%)	Producer (%)	Solver (%)
R1	100	100	100
R2	100	100	100
R3	100	100	100
R4	89.8	100	100
R5	69.9	90.0	100
R6	50.0	60.0	79.9
R7	29.6	48.9	50.0
R8	9.8	20.0	50.0

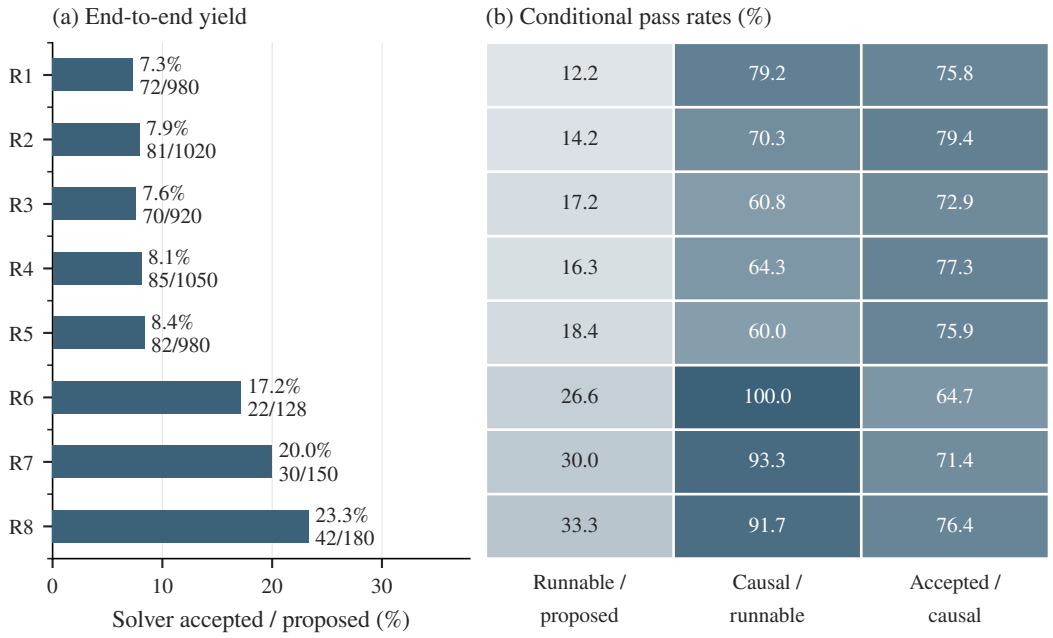


Figure 4: Production yield and stage-specific acceptance. (a) Accepted tasks as a fraction of proposals; labels give both the percentage and its counts. (b) Each stage's pass count divided by its preceding stage's count, in percent; the product of these fractions equals the end-to-end yield. All eight requests are included; Table 4 gives the counts.

C.1 CONTROLLED ROLE EVALUATIONS

Tables 6–8 report the controlled evaluations described in Section 4.3. Critic judgments use two independent, model-blinded human raters and third-rater adjudication. Correct-and-actionable feedback is a subset of correct diagnoses. Producer validity requires executable tests, the original-pass/changed-fail/reverted-pass check, blinded target-gap matching, and within-model deduplication. Its 200 attempts comprise four proposals for each of 50 shared gap/repository conditions. In the feedback experiment, the fixed R8 Solver makes one retry per failure state under each feedback condition.

Table 6: Critic quality on 200 fixed failure traces excluded from training and adaptive feedback. Both rates use all 200 traces as the denominator.

Model	Correct	Correct + actionable	Accuracy (%)	Actionable (%)
M_0 (R1)	78	49	39.0	24.5
R4	111	82	55.5	41.0
R6	133	108	66.5	54.0
R8	157	137	78.5	68.5
Teacher	174	158	87.0	79.0

Table 7: Producer quality from 200 attempts per model on 50 fixed conditions. Valid tasks additionally match the target gap and are unique within the model's output. Percentages use all attempts as the denominator.

Model	Runnable	Causal pass	Valid + unique	Runnable (%)	Valid (%)
M_0 (R1)	43	25	14	21.5	7.0
R4	61	42	27	30.5	13.5
R6	80	61	42	40.0	21.0
R8	103	84	63	51.5	31.5
Teacher	131	113	91	65.5	45.5

Table 8: Feedback utility on the same 200 failed tasks with fixed R8 Solver and retry budgets. Differences are marginal success-rate differences.

Feedback	Repaired/200	Rate (%)	Change vs. none (pp)
None	38	19.0	—
M_0 Critic	46	23.0	+4.0
Self-Refine-style	52	26.0	+7.0
R8 Critic	67	33.5	+14.5

Relative to no feedback, M_0 Critic, Self-Refine-style, and R8 Critic feedback yield 10, 16, and 31 newly successful tasks, respectively, with two regressions in each condition. Their net gains are therefore 8, 14, and 29 tasks, consistent with Eq. equation 5.

D ROLE AND SUPERVISION ACCOUNTING

The R6–R8 Solver counts are 274, 380, and 520; the corresponding all-role final-training totals are 334, 533, and 712. Historical R6 counts of 98 Critic and 34 Producer records are distinct from the 40 and 20 records in final training (Table 12).

The R1–R5 release-task counts exceed the Solver-acceptance counts in Table 4. Release tasks and accepted proposals are distinct reporting units.

For R6, the separate role-token entries are 35,000 Critic, 28,000 Producer, and 242,742 Solver tokens, totaling 305,742. The round entry of 242,742 covers only Solver tokens. R1–R5 sum to 8.8M tokens; adding the R6 Solver entry gives 9.042742M before R7. This subtotal rounds to 9.04M.

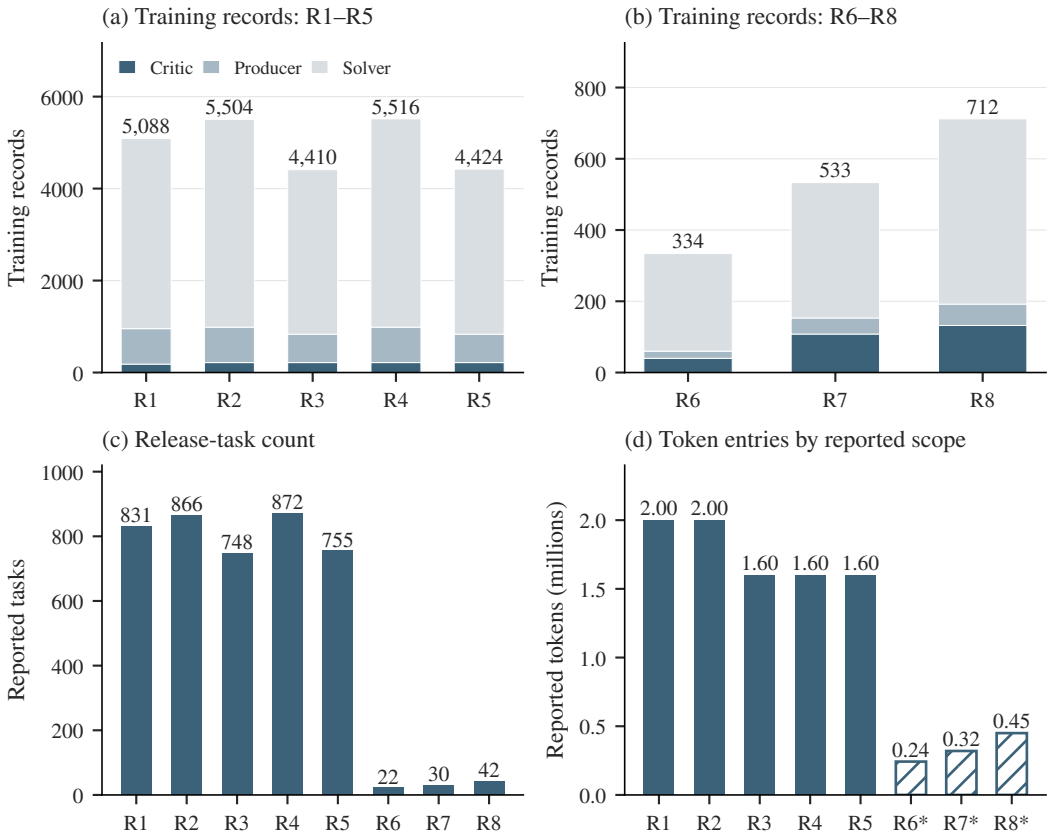


Figure 5: Training data by round. (a,b) Final-training role counts and their sums; vertical scales differ between R1–R5 and R6–R8. (c) Release-task counts, recorded separately from Solver acceptance. (d) Token entries in millions, rounded to two decimals. Hatching and asterisks identify R6 Solver-only tokens and R7–R8 entries with unconfirmed all-role coverage. Tables 9 and 11 give exact values.

Table 9: Final-training record counts, counting each record once regardless of epochs. The final column sums the three roles.

Request	Critic	Producer	Solver	Total
R1	183	771	4,134	5,088
R2	216	771	4,517	5,504
R3	216	618	3,576	4,410
R4	216	771	4,529	5,516
R5	216	618	3,590	4,424
R6	40	20	274	334
R7	108	45	380	533
R8	132	60	520	712

Table 10: Final-training source counts. S denotes independently generated Student records; T denotes Teacher-generated or Teacher-rewritten records. Mixed and unknown source counts are zero in every row. S + T gives the role total in Table 9; epochs do not multiply counts.

	Critic		Producer		Solver	
Request	S	T	S	T	S	T
R1	0	183	0	771	0	4,134
R2	0	216	0	771	0	4,517
R3	0	216	0	618	0	3,576
R4	22	194	0	771	0	4,529
R5	65	151	62	556	0	3,590
R6	20	20	8	12	55	219
R7	76	32	23	22	190	190
R8	119	13	48	12	260	260

Table 11: Reported release tasks and supervision-token entries, with each entry's accounting scope.

Request	Release tasks	Reported tokens	Scope
R1	831	2,000,000	Supplied round total
R2	866	2,000,000	Supplied round total
R3	748	1,600,000	Supplied round total
R4	872	1,600,000	Supplied round total
R5	755	1,600,000	Supplied round total
R6	22	242,742	Also identified as Solver tokens
R7	30	320,000	Round/family coverage unresolved
R8	42	450,000	All-role coverage not specified
R1–R5 sum	—	8,800,000	Arithmetic sum of listed token entries

Table 12: R6 historical role counts, final-training counts, and token entries. The 242,742 Solver tokens also appear in Table 11. Release and filtering-stage alignment of the two count series is unconfirmed.

Role	Historical count	Final-training count	Token entry
Critic	98	40	35,000
Producer	34	20	28,000
Solver	274	274	242,742
Arithmetic sum	406	334	305,742

E RESOURCE REPORTING

Table 13: Stage-specific API estimates; K denotes 1,000 tokens. The final row sums the charge thresholds, not actual or end-to-end costs.

Reported stage	Input tokens	Output tokens	Estimated charge
R4 feedback	72K	45K	$< \$0.05$
R5 feedback	63K	25K	$< \$0.04$
R6 Critic	705K	69K	$< \$0.10$
R6 Solver	7,465K	456K	$< \$1.00$
R7 student-assisted	420K	40K	$< \$0.05$
R8 student-driven	180K	20K	$< \$0.02$
Sum of listed thresholds	—	—	$1.26

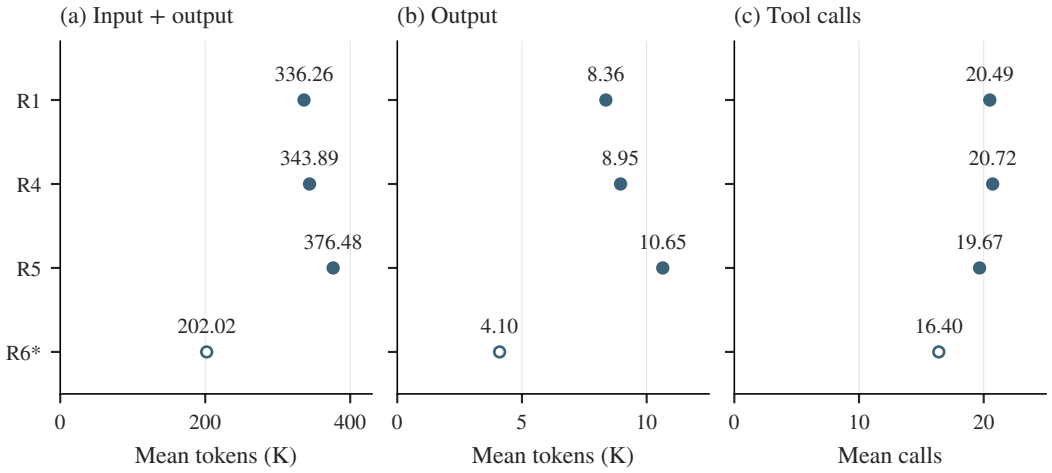


Figure 6: Reported per-task inference means: (a) input plus output tokens, (b) output tokens, and (c) tool calls. Token axes use thousands (K). Hollow markers and an asterisk identify provisional R6 values. Sample denominators and alignment with the 500-task evaluation are unconfirmed. Exact means appear in Table 14.

Table 14: Reported per-task inference averages. Sample denominators and alignment with the 500-task evaluation are unconfirmed. R6 values are provisional.

Request	Input + output tokens	Output tokens	Tool calls
R1	336,262.00	8,363.80	20.49
R4	343,891.93	8,954.59	20.72
R5	376,475.13	10,646.00	19.67
R6 (provisional)	202,016.60	4,103.20	16.40

F ILLUSTRATIVE TRACE-TO-TASK EXAMPLE

The conceptual example in Figure 1 follows execution feedback through the three roles. Its task specification requires `normalize(x)` to return `0.0` when `x == 0`, and `100.0 / x` otherwise. The example starts with an unguarded division and a visible `ZeroDivisionError` on the zero-input case.

Critic input and output. The Critic reads the task specification, the student's attempted patch, and the associated test trace. It diagnoses missing zero-input handling from the visible exception.

Producer construction. The Producer constructs boundary-case practice with focused checks: `normalize(0) == 0.0` and `normalize(4) == 25.0`. The constructed task is subject to the admission protocol.

Solver candidate. A candidate implementation is:

```
def normalize(x):
    if x == 0:
        return 0.0
    return 100.0 / x
```

The Solver receives the specification and defective workspace without a reference implementation. External execution of the specified checks determines acceptance. The figure's test-pass count and runtime are illustrative.