PATCHES HIDE TESTS TO WRITE: DERIVING TRAINABLE TEST-GENERATION TASKS FROM EXISTING SWE ENVIRONMENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Reinforcement learning for repository-level coding agents is limited by the supply of executable tasks, and the work that enlarges it either curates more real issues or manufactures new defects. An issue-resolution instance carries four artifacts: the problem description, the repository snapshot in which the issue is present, the reference patch that fixes it, and the reference tests that the fix turns from failing to passing. Issue resolution withholds the patch and judges with the tests. Exchange the two roles and the same instance defines a second task, writing a test that reproduces the reported behaviour, inside the environment the instance already ships and at no annotation cost. Fail-to-pass status alone is satisfied by tests that exercise no behaviour, such as one that checks whether a line of the buggy source is absent. We therefore require a submitted test to fail both on the buggy snapshot and on snapshots in which the reference patch has been damaged, and to pass both on the repaired snapshot and on versions of it whose source text has been rewritten without changing behaviour. Applied to three collections of instances that differ in where their defects and tests come from, the conversion produces 30,605 fail-to-pass test-generation tasks, of which 23,520 pass an audit of their verifiers. Forty steps of joint reinforcement learning on the two tasks, which draw about 480 tasks in total, raise Qwen3.5-9B from 48.2% to 55.0% on SWE-bench Verified and from 38.5% to 44.5% on TDD-Bench Verified, ahead of training on either task alone. We release SWE-Flip, the conversion pipeline together with the tasks it produced.

1 INTRODUCTION

Reinforcement learning has become the standard means of improving a coding agent on repository-level work, and its principal constraint is the supply of tasks available for training. A usable task requires a repository, an environment in which its tests execute, and a reward that execution can compute. SWE-bench assembled such instances from real GitHub issues (Jimenez et al., 2024), and subsequent work has enlarged the supply, either by curating further issues and shipping them with runtimes (Pan et al., 2024; Badertdinov et al., 2025; Zhao et al., 2026) or, where real issues are exhausted, by manufacturing the defect (Jain et al., 2025; Yang et al., 2025a; Sonwane et al., 2025). In each case the addition is an instance of the same task: the agent is given a problem description and a repository in which some behaviour is broken, and must produce a patch.

An instance of that task carries four artifacts: the problem description, the repository snapshot in which the issue is present, the reference patch that resolves it, and the reference tests that the patch turns from failing to passing. Issue resolution withholds the patch from the agent and judges its submission with the tests, but that division of the four artifacts is not the only admissible one. Under the opposite division, the agent receives the same description and the same snapshot, the tests are withheld in place of the patch, and the submission required is a test that reproduces the reported behaviour, judged by whether it fails before the reference patch is applied and passes afterwards. This is the fail-to-pass task that SWT-Bench and TDD-Bench Verified evaluate (Mündler et al., 2024; Ahmed et al., 2024), and obtaining it from a resolution instance costs no annotation beyond what that instance already carries. Figure 1 shows the two tasks together with the checks that decide each reward.

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.

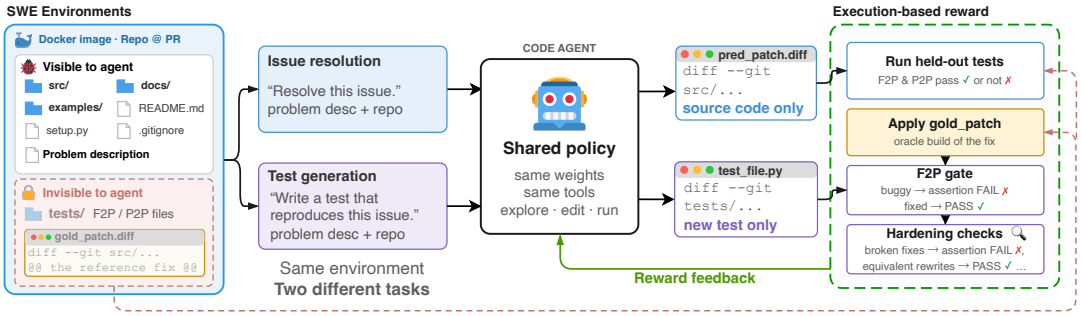

Figure 1: One SWE environment, two tasks, and the checks that decide each reward. The agent sees the problem description and the repository in its buggy state; the reference tests and the reference patch are withheld and reach only the grader. The same policy is asked either to resolve the issue, submitting a patch to the source code, or to write a test that reproduces it, submitting a new test file. A patch is rewarded when the withheld tests pass. A test is rewarded when it fails on the buggy repository and passes once the reference patch is applied, and when it also survives the hardening checks of Section 3.3: it must still fail when the reference patch is damaged, and still pass when the repaired source is rewritten without changing its behaviour.

Building the second task takes no annotation: a script removes the reference tests from the image the agent works in, scrubs the repository history that would reveal the later commit, and fixes the path a submitted test must be written to. Supplying a reward that is safe to train on is the part that takes work. The fail-to-pass criterion requires only that a submitted test fail on the buggy snapshot and pass on the repaired one, and a test can satisfy both conditions without exercising any behaviour: it may assert that a particular line of the buggy source is absent, or compute a fingerprint of the abstract syntax tree of the file the patch modifies. Such a submission collects the reward of a correct test while testing nothing, and the criterion does not separate it from a test that reproduces the issue. That execution-based criteria admit submissions of this kind is not specific to us: an audit of 49 tasks in SWE-bench Verified finds that 28.5% of the tasks it examines accept a patch that does not fix the issue (Rajan, 2026). We therefore impose a stricter criterion, defined against several versions of the repository on each side rather than one. A submitted test must fail on the buggy snapshot and also on snapshots in which the reference patch has been damaged; it must pass on the repaired snapshot and also on versions of that snapshot whose source text has been rewritten without changing behaviour. A test keyed to source text or to parse structure fails one of these two requirements.

We apply the conversion to three collections of instances that differ in where their defects and their tests originate, which separates the cost of the conversion itself from what depends on the source. It produces 30,605 test-generation tasks, of which 23,520 are admitted. Admission audits each verifier against tests whose correct verdict is known in advance: the reference tests, which it must reward, and, on a sample, tests of the kind described above, which it must refuse. Because both tasks are defined on the same issue, one collection supplies both.

Training on these tasks improves the agent at both of them. Forty steps of joint reinforcement learning, which draw about 480 tasks, raise Qwen3.5-9B from 48.2% to 55.0% on SWE-bench Verified and from 38.5% to 44.5% on TDD-Bench Verified, ahead of training on either task alone. The converted tasks also serve as supervised data, since the same verifiers decide which collected trajectories are worth imitating (Section 5.3).

Our contributions can be summarized as follows:

- We propose a conversion pipeline that turns an issue-resolution instance into a fail-to-pass test-generation task inside the environment the instance already ships, and that rewards the task under a criterion stricter than fail-to-pass status alone: we require a submitted test to fail as well when the reference patch is damaged, and to pass as well when the repaired source is rewritten without changing its behaviour, so that a test with no behavioural content is refused.

- We release 30,605 converted tasks from three sources, 23,520 of which we admit after auditing their verifiers, together with the pipeline itself and the per-stage counts that show what each source costs and what limits it.
- We find that joint reinforcement learning on resolution and test generation improves both tasks, and improves each beyond what we obtain by training on that task alone; we also use the same tasks as supervised data.

2 Related Work

Executable environments for software-engineering agents. SWE-bench pairs a real GitHub issue with the tests its fix turns from failing to passing, and the scaffolds built on it made issue resolution the standard task for a coding agent (Jimenez et al., 2024; Yang et al., 2024; Xia et al., 2024; Wang et al., 2024); SWE-bench Pro raises the difficulty with long-horizon problems, its public set having 731 instances (Deng et al., 2025). Training consumes such instances faster than evaluation, so recent work enlarges the supply, either by curating issues and shipping them with runtimes (Pan et al., 2024; Badertdinov et al., 2025; Zhang et al., 2025; Zhao et al., 2026; Shen et al., 2026) or by manufacturing the defect: SelfAPR perturbs earlier versions of the program (Ye et al., 2022), R2E-Gym back-translates commits and generates both the tests and the issue text (Jain et al., 2025), SWE-smith injects defects into Python repositories (Yang et al., 2025a), and BugPilot has an agent implement a feature that breaks existing tests (Sonwane et al., 2025). SWE-Dev synthesises the fail-to-pass tests themselves (Wang et al., 2025a), and SWE-Playground generates projects from scratch and trains one agent on resolution and reproduction together (Zhu et al., 2025). Almost all of this work adds instances of one task, and where a second task appears it arrives with new material to build. We add a test-generation task to the environments that already exist.

Reinforcement learning for software engineering. Reinforcement learning on repository tasks takes its reward from the human patch or from the repository's tests. SWE-RL scores a candidate patch by its similarity to the human patch, without execution (Wei et al., 2025); the execution-based alternative runs the tests inside the container (Golubev et al., 2025; FAIR CodeGen team et al., 2025). Both rewards are sparse at the scale of an episode, and much of the work since addresses that: by supplying plans and error feedback (Da et al., 2025), by rewarding the sub-skills of a workflow separately (Ma et al., 2025; Yang et al., 2025b), by replacing the test outcome with a learned reward model (Shum et al., 2025), or by compacting the agent's context (Li et al., 2026). Self-play SWE-RL needs neither human issues nor human tests: one model injects a defect and repairs it (Wei et al., 2026). Most of these runs use GRPO (Shao et al., 2024) or its dynamic-sampling variant DAPO (Yu et al., 2025).

Generating a test from an unresolved issue. SWT-Bench turns the instances of an issue-resolution benchmark around: the agent sees the issue and the buggy repository and must produce a test that fails there and passes once the reference fix is applied (Mündler et al., 2024). TDD-Bench Verified supplies 449 human-filtered issues and grades a submission on fail-to-pass status together with its coverage of the changed code (Ahmed et al., 2024); the same task has since been posed for Java (Ahmed et al., 2026). Reported reproduction rates come from staged inference (Wang et al., 2025b; Ahmed et al., 2025a;b; Nashid et al., 2025; Khatib et al., 2025; Wang et al., 2026b). Such tests also rank candidate patches, doubling the precision of SWE-agent in SWT-Bench and selecting fixes in an industrial code base (Cheng et al., 2025). Training for the task is rarer: the tests an agent writes on its way to a patch serve mostly as print statements, and prompting it to write more does not change how many issues it resolves (Chen et al., 2026). Soni et al. (2026) assemble 41K reproduction instances from 2.6K repositories and train open-weight models on them. At function level a code generator and a test generator are trained against each other, rewarded by agreement rather than by execution against a human fix (Lin et al., 2025; Wang et al., 2025c; 2026a); UTRL rewards a test for passing the reference solution while failing a solution the code generator produced (Lee et al., 2026), TCS restricts its second stage to the solver's current failure modes (Xu et al., 2026), and closest to the reward we define, RobustTests synthesises tests against deliberately near-correct faulty programs rather than against correct ones (Zhang et al., 2026). R2E-Gym generates tests only as a means of building resolution environments (Jain et al., 2025). We convert issue-resolution instances instead, so that every test-generation task is paired with the resolution task it came from and both

are drawn from the same issues. Exchanging input and supervision to multiply data is an old idea, as in dual learning for translation (Xia et al., 2016); here both directions are already executable.

3 Pairing Test Generation with Issue Resolution in SWE Environments

3.1 Task Formulation

An instance of an SWE environment provides four artifacts: a problem description, a repository snapshot S in which the issue is present, a reference patch $P^\star$, and reference tests $T^\star$ that the fix turns from failing to passing. Write $S \oplus P$ for the snapshot with a patch P applied and $S^\star = S \oplus P^\star$ for the repaired snapshot, and let

$$\mathrm{out}(t, S) \in \{\textsf{pass}, \textsf{fail}, \textsf{error}\} \tag{1}$$

be the result of executing a test file t on a snapshot S, read from the tests that ran rather than from the exit status. Writing $(n_\mathrm{p}, n_\mathrm{f}, n_\mathrm{e})$ for the number of tests in t that passed, failed and errored, the outcome is fail when $n_\mathrm{f} \geq 1$, pass when $n_\mathrm{f} = n_\mathrm{e} = 0$ and $n_\mathrm{p} \geq 1$, and error otherwise. Every test in the file must therefore pass for the outcome to be pass, and at least one must have run: a file whose tests are all skipped satisfies the first requirement but not the second, and it exits zero, which is why the exit status is not what the reward is read from. A collection error, an import error and a timeout are error as well.

Issue resolution asks for a patch P. Test generation keeps the problem description and the snapshot and asks instead for a fail-to-pass (F2P) test (Mündler et al., 2024; Ahmed et al., 2024): the agent writes a single test file t at a fixed path, and nothing else in the repository may differ when the episode ends. The reward for a patch, and the criterion the benchmarks apply to a test, are

$$r_\mathrm{fix}(P) = \mathbf{1}\left[\mathrm{out}(T^\star, S \oplus P) = \textsf{pass}\right], \qquad \mathrm{F2P}(t) = \mathbf{1}\left[\mathrm{out}(t, S) = \textsf{fail} \ \wedge \ \mathrm{out}(t, S^\star) = \textsf{pass}\right]. \tag{2}$$

The two tasks thus differ in which artifact is withheld and which one judges. In issue resolution the reference patch is the withheld answer and the reference tests are the judge; in test generation the reference tests are a withheld reference and the reference patch is the judge. Neither artifact is available to the agent in either task: graded material is mounted only at scoring time, the container has no network access, and repository history that would expose the later commit is removed. Appendix A lists the measures in full. Because both tasks are defined on the same issue and the same snapshot, an instance that converts yields a pair of tasks, and the second costs no annotation beyond what the resolution instance already carried.

3.2 Task Conversion

A script performs the construction, and it is the pipeline we release as SWE-Flip. The issue text becomes the prompt; the agent's image is built from the same snapshot with the reference tests deleted and the repository history scrubbed; the submission path is fixed in advance, so that the verifier knows what to execute and the agent cannot satisfy the reward by editing tests that already exist; the reference patch and the reference tests are held outside that image and mounted only when a submission is scored; and the verifier is written per task. Each task is emitted in the format of the Harbor harness (Harbor Framework Team, 2026). Nothing in the construction requires the task to be a repository issue; Appendix D applies it to a collection of terminal tasks.

Not every instance defines a solvable test-generation task, and the conditions can be checked without model rollouts. We retain an instance when the reference patch modifies library source rather than tests, examples or benchmarks, so that there is behaviour to observe; when the reference tests fail on the buggy snapshot through a failing assertion rather than a collection error, since an error usually indicates that the test imports a symbol introduced by the patch, which an agent that cannot see the fix is unable to reference; when the issue text is specific enough to identify the behaviour at stake; when the failure is deterministic; and when the problem description and the reference patch concern a single behaviour. The last condition carries a different consequence from the others: a patch that bundles unrelated edits makes the verifier demand a test for a change the description never mentions, so a test written correctly from the description is scored wrong, and the reward becomes misleading

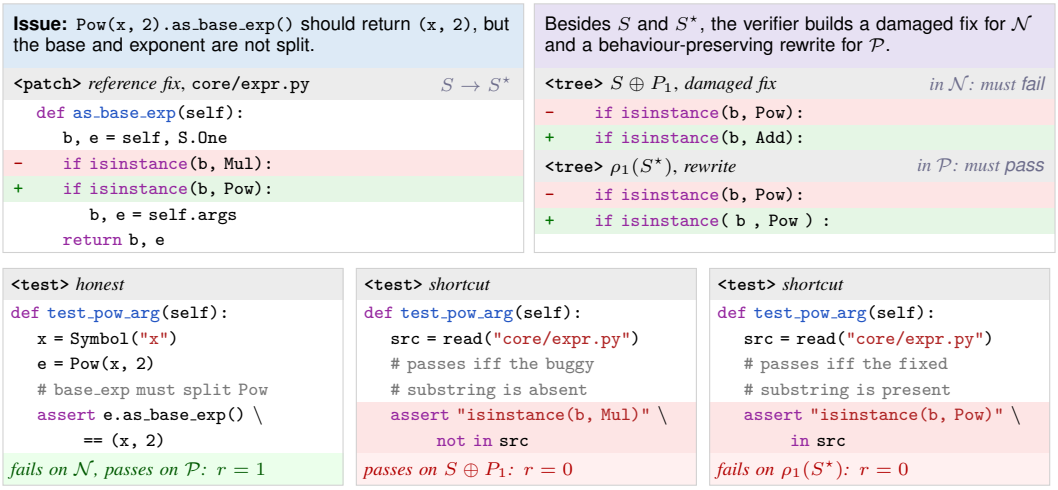

Figure 2: The two added snapshots on one instance. All three tests pass the fail-to-pass check; each shortcut is refused by one added snapshot.

rather than merely unavailable. Such instances are removed at conversion time, and weakening the verifier does not recover them, since nothing in the task identifies which part of the patch the description covered.

3.3 Task Verification

The fail-to-pass verification defines the reward, but it does not bound what can collect it. In issue resolution the agent submits source code, and the relevant question about a task is whether its hidden tests are sufficiently strict. Here the agent submits test code that the verifier subsequently executes, so a submission can satisfy both runs without exercising the behaviour in question: it can assert the absence of a string from the buggy source, fingerprint the syntax tree of the modified file, or read state left behind by the verifier on an earlier run. Such a submission has no behavioural content, and the fail-to-pass verification cannot distinguish it from an honest test.

We therefore collect two families of repository snapshots. The family $\mathcal{N}$ holds snapshots whose behaviour on the issue differs from that of the repaired one: the buggy snapshot S, and $S \oplus P_i$ for damaged versions $P_1, \ldots, P_m$ of the reference patch. The family $\mathcal{P}$ holds snapshots whose behaviour is identical to it: the repaired snapshot $S^\star$, and $\rho_j(S^\star)$ for rewrites $\rho_1, \ldots, \rho_n$ that change its source text but not its behaviour. That is,

$$\mathcal{N} = \{S\} \cup \{S \oplus P_i\}_{i=1}^{m}, \qquad \mathcal{P} = \{S^\star\} \cup \{\rho_j(S^\star)\}_{j=1}^{n}, \tag{3}$$

and the verifier rewards a submission only when it separates the two families,

$$r_{\text{test}}(t) = \begin{cases} 1, & \text{out}(t, X) = \textsf{fail} \text{ for all } X \in \mathcal{N} \text{ and } \text{out}(t, Y) = \textsf{pass} \text{ for all } Y \in \mathcal{P}, \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

The criterion F2P of Equation 2 is the case $\mathcal{N} = \{S\}$, $\mathcal{P} = \{S^\star\}$; the snapshots we add to either family are what a test with no behavioural content fails to survive, as Figure 2 shows on one instance. Requiring fail rather than merely the absence of pass on $\mathcal{N}$ carries its own weight: a test that names a symbol the fix introduces raises an import error on S, which is an error and does not count. What remains is how the two families are filled.

- **$\mathcal{N}$ is filled by damaging the reference patch, and the reference tests decide which damage counts.** The verifier reverts or corrupts individual parts of $P^\star$ and keeps a variant only when the reference tests fail on it. A variant they still pass has not changed observable behaviour, so it is dropped rather than required of the agent, which is why the damage need not be applied carefully.
- **$\mathcal{P}$ is filled by rewriting the repaired snapshot, and equivalence is decided on the syntax tree.** A rewrite ρ_j is used only when the files it produces parse to the tree the originals

Table 1: The conversion applied to each source. *Reproduce F2P*: instances whose reference tests fail on the buggy snapshot and pass on the repaired one. *Converted*: test-generation tasks produced from them. *Admitted*: tasks whose verifier ran every trial and rewarded the reference test (Section 4.2). *Repositories*: distinct repositories covered by the converted and by the admitted tasks.

Source	Instances	Reproduce F2P	Converted	Admitted	Repositories
Scale-SWE	19,473	17,470	8,243	6,956	953 / 888
SWE-smith-py	31,873	21,441	14,352	13,250	112 / 107
R2E-Gym	8,095	8,049	8,010	3,314	13 / 13

parse to, compared after normalisation and while ignoring inserted no-operations, and when every scope binds the same names. Renaming local variables and normalising the tree is not enough on its own, because unparsing returns many lines byte for byte and a test asserting that a line of the patch appears in the source would still pass; the rewrites therefore also insert spaces between tokens and split string literals into adjacent literals that the compiler joins again. Which rewrites are applied, and in which order, is drawn per task, since a fixed sequence is itself a regularity a test can be written against.

One requirement is not a condition on a snapshot and therefore not part of Equation 4. The trials run in one container, one after another, so a test that can tell which trial it is in can answer each one as expected without testing anything. We clear writable state between trials, keep the verifier's own output off paths the test can read, and draw the order of the trials per task. The verifier also runs the same test twice on trials it cannot tell apart and requires the same verdict both times.

4 Properties of the Converted Tasks

We apply SWE-Flip to three collections of issue-resolution instances that differ in where the defect and the tests come from. Scale-SWE (Zhao et al., 2026) takes the fix from a merged pull request and has an agent pipeline supply the environment, the tests and the problem description. SWE-smith (Yang et al., 2025a) injects defects into repositories whose tests were passing; we use its 31,873 Python instances and call them SWE-smith-py. R2E-Gym (Jain et al., 2025) back-translates commits and generates both the tests and the problem description. Comparing the three separates what the conversion costs from what depends on the source.

4.1 Scale and Composition

Conversion loses instances for one main reason, visible before any container is run: the reference patch and the problem description do not concern the same behaviour. Scale-SWE converts just under half of the instances that reproduce, and five in six of the rest fail the single-behaviour condition of Section 3, because a real commit often bundles edits its description never mentions. In SWE-smith-py the mismatch is built into some generators, which combine several injected edits into one defect or have a model rewrite a whole function while the description covers one change. These generators produce 30% of the source and we exclude them before running any test. R2E-Gym converts over 99% of the instances that reproduce; what it loses, it loses at admission rather than at conversion. Appendix B gives the counts behind each step.

Together the three sources give 23,520 admitted tasks, and they differ in what they contribute. Scale-SWE contributes breadth, with 888 repositories; SWE-smith-py contributes volume, more than half of the admitted tasks; R2E-Gym contributes concentration, 3,314 tasks in 13 repositories. Because both tasks are defined on the same issue, one collection trains both.

4.2 Admission

Before training, each converted task is run once with its reference tests in place of the agent's test, and it is admitted only if the verifier rewards them and every trial of Equation 4 was actually carried out. The second condition is needed because some tasks have no damaged version of the reference

patch to offer. On those a test earns the full reward without ever being confronted with one. Appendix C reports how often the verifier refuses the shortcuts of Section 3.3 and on how many probe runs. The same principle applies to the other task: a resolution task is admitted when the reference tests fail on at least one damaged version of the reference patch through a failing assertion rather than through a crash, which holds for 4,677 of the 19,473 Scale-SWE instances.

A verifier is only as discriminating as the reference tests that decide which damaged fixes count, and this is where the sources differ most. Admission keeps 84% of the converted Scale-SWE tasks and 92% of the SWE-smith-py tasks, but only 41% of the R2E-Gym tasks. Nearly half of the R2E-Gym losses, 2,169 of 4,696, are tasks on which no damaged version of the reference patch makes a fail-to-pass reference test fail, so $\mathcal{N}$ holds only the buggy snapshot and the trial against damaged fixes cannot be run: those tests react to the fix as a whole but not to parts of it taken away. The SWE-smith-py rate is higher partly by selection, since we converted only instances whose reference tests had already been shown to reject damaged versions of the fix on the resolution side. What limits how much of a source can be trained on is the strength of its reference tests rather than the number of its tasks.

5 Experiments

5.1 Setup

We train Qwen3.5-9B (Qwen Team, 2026a) with GRPO (Shao et al., 2024), using mini-swe-agent (Yang et al., 2024) as the scaffold with interleaved thinking enabled. At each step we draw 12 tasks and sample 8 trajectories for each at temperature 1.0; a group in which every trajectory receives the same reward has no advantage and is discarded, and 4 of the remaining groups, 32 trajectories, form the update. We use a clip ratio of 0.2, with no KL or entropy regularisation, and Adam with a constant learning rate of 10^{-6} and weight decay 0.1. Each model is trained for 40 steps, which draws about 480 tasks, a small fraction of the admitted Scale-SWE tasks. Each training trajectory is capped at 80 turns and a context of 65,536 tokens, and each run uses 4 machines of 8 H800 accelerators.

Training tasks come from Scale-SWE, filtered so that a group is likely to produce a gradient. On the resolution side we keep only tasks whose reference patch is between 1,287 and 3,982 bytes: a smaller patch tends to be solved in all eight trajectories and a larger one in none, and either way the group is discarded. Patch size does not predict the difficulty of a test-generation task, which is left unfiltered. From both sides we drop the tasks that earlier runs solved in no trajectory at all. This leaves 1,329 resolution and 973 test-generation tasks; the latter side is deliberately smaller, since resolution groups are discarded more often and drawing fewer test-generation tasks equalises what reaches the update.

We evaluate with a cap of 250 turns and a context of 262,144 tokens on SWE-bench Verified (Jimenez et al., 2024), the 731 public instances of SWE-bench Pro (Deng et al., 2025) and TDD-Bench Verified (Ahmed et al., 2024). The tasks drawn in the 40 steps share no instance with them.

5.2 Joint Training on Resolution and Test Generation

The three trained models differ only in the tasks they are trained on: each single-task model draws from its own part of the tasks of Section 5, the joint model from both. Discarding the groups without reward variance removes 53% of the resolution groups but 39% of the test-generation groups.

Table 2 gives the results. The joint model is 6.8 points above the base model on SWE-bench Verified and 6.0 on TDD-Bench Verified, and it is the best of the three trained models on every benchmark. Training on resolution alone leaves SWE-bench Verified almost unchanged and falls below the base model on SWE-bench Pro and TDD-Bench Verified. Training on test generation alone raises SWE-bench Verified by 2.2 points and TDD-Bench Verified by 1.8, although it is only ever rewarded for tests, never for a patch.

The behaviour counts of Table 5 account for the decline of the resolution-only model: training on that task alone removes testing from the trajectory. Its reward is paid for the patch alone, and after

Table 2: Success rate (%) on three benchmarks: issues resolved on the two SWE-bench sets, tests that reach fail-to-pass on TDD-Bench Verified. All models start from Qwen3.5-9B and are trained for 40 steps. The TDD score row gives the coverage-weighted score of TDD-Bench Verified (Ahmed et al., 2024) on the same runs.

		Trained on one task		
	Base	Issue resolution	Test generation	Joint
SWE-bench Verified	48.2	48.4	50.4	**55.0**
SWE-bench Pro	25.0	20.8	23.9	**26.0**
TDD-Bench Verified	38.5	35.0	40.3	**44.5**
– TDD score (0–1)	0.355	0.323	0.374	**0.411**

Table 3: Tests that reach fail-to-pass (%) on TDD-Bench Verified. The second column drops the four instances that the fine-tuning data also contains, and the third drops every instance from the three repositories the two share, `sympy`, `matplotlib` and `flask`. We use the `claude-code` scaffold in the SFT experiment.

TDD-Bench Verified	All (449)	Without the 4 shared instances (445)	Without the 3 shared repositories (349)
Qwen3.5-9B	26.1	25.8	22.6
– SFT fine-tuned	73.3	73.0	73.4
GLM-5.3	87.8	87.6	87.7
Qwen3.8-Max	86.9	86.7	86.3

40 steps the model writes a test into the repository on 5.2% of the instances of SWE-bench Verified rather than 11.2%, and ends 41.4% rather than 25.1% without having run a test. The two benchmarks on which it falls below the base model depend on that behaviour: TDD-Bench Verified takes a test as the submission, and the long-horizon issues of SWE-bench Pro require an edit to be checked by execution before it is submitted. Joint training moves the same counts in the opposite direction, and the model with the highest rate of test writing also has the highest resolved rate.

What the joint model fails at is not finding the code the issue concerns. On TDD-Bench Verified its tests execute the lines the reference patch changed about equally well whether or not they reach fail-to-pass: the median share executed is 100% in both cases, and the two differ only in the lower quartile, 100% against 50%. Coverage of the change therefore does not separate a test that reproduces the issue from one that does not; what binds is the behaviour required on the two sides. We count an instance whose coverage report is absent as covering nothing, where the reference implementation counts it as covering everything.

5.3 Supervised Fine-Tuning on Verified Trajectories

The converted tasks are also a source of supervised data, and here all three contribute. We sample test-generation trajectories from GLM-5.3 (Z.ai, 2026) and retain those whose submitted test the verifier of Equation 4 rewards. This leaves 8,418 trajectories — 2,834 from SWE-smith-py, 2,798 from R2E-Gym and 2,786 from Scale-SWE — which hold 0.63B tokens, 75,000 per trajectory, and run for 28 turns and 32 tool calls. Fine-tuning the base model of Section 5 on them, with the learning rate decayed from 5×10^{-5} to 5×10^{-6}, raises TDD-Bench Verified from 26.1% to 73.3%, against 87.8% for the teacher (Table 3).

Part of the fine-tuning data comes from repositories TDD-Bench Verified also draws on, so Table 3 reports the benchmark under two exclusions. Four of its 449 instances appear in the fine-tuning data, and excluding them gives 73.0%; excluding the three shared repositories altogether leaves 349 instances and gives 73.4%. GLM-5.3 and Qwen3.8-Max (Qwen Team, 2026b) shift by at most 0.6 points under the same exclusions, and the base model falls from 26.1% to 22.6%, so the exclusions leave the harder instances rather than the easier, and the 73.4% is not an artifact of what they remove.

Table 4: The 449 issues of TDD-Bench Verified, which are instances of SWE-bench Verified, classified at once by whether the model resolved the issue and by whether the test it wrote for that same issue was rewarded.

	Base		Joint	
	Test rewarded	Test not rewarded	Test rewarded	Test not rewarded
Issue resolved	113	105	144	107
Issue unresolved	60	171	56	142

Table 5: What the agent does on the 500 instances of SWE-bench Verified, counted from its trajectories.

		Trained on one task		
	Base	Issue resolution	Test generation	Joint
Commands per model turn	0.60	0.50	0.54	1.00
Tasks with a test written under `tests/`	11.2%	5.2%	5.8%	18.2%
Test runs per task (median)	9	5	7	11
Tasks with no test run	25.1%	41.4%	27.9%	10.0%
Tasks that reach the 250-turn limit	28.7%	13.4%	18.2%	14.8%

5.4 The Two Tasks on the Same Issue

TDD-Bench Verified is built from the instances of SWE-bench Verified, so the same model is asked both to write a reproducing test and to fix the issue. The corresponding rows of Table 2 are therefore two measurements on one set of issues, and what one task does for the other can be read off them.

Table 4 classifies those issues by both outcomes at once. The base model resolves 218 of the 449 issues and earns a reward for the test it writes on 173 of them, and 113 issues fall in both groups. Joint training raises all three counts, to 251, 200 and 144. The two capabilities move together across issues rather than trading off against each other.

Table 5 counts what the agent does while resolving the 500 instances of SWE-bench Verified, and four of those counts move together: training on either task alone moves the agent away from testing, and training on both moves it towards testing. The joint model writes a test into the repository's `tests/` directory on 18.2% of the instances against 11.2% before training, runs a median of 11 tests per instance against 9, ends 10.0% of the instances rather than 25.1% without having run a test, and issues 1.00 commands per model turn against 0.60. Both single-task models move all four counts in the opposite direction.

6 Conclusion

We presented a pipeline that converts an issue-resolution instance into a fail-to-pass test-generation task. The conversion withholds the instance's reference tests in place of its reference patch and needs no new annotation; what it does need is a criterion strict enough to refuse a test with no behavioural content, so the verifier judges a submission on damaged versions of the reference patch and on behaviour-preserving rewrites, not on the buggy and repaired snapshots alone. Applied to three collections of instances it produced 30,605 tasks, of which 23,520 were admitted, and we release these as SWE-Flip with the pipeline that built them. Forty steps of joint reinforcement learning took Qwen3.5-9B from 48.2% to 55.0% on SWE-bench Verified and from 38.5% to 44.5% on TDD-Bench Verified, past either task alone. We leave as future work checking the agent's patches with the agent's own tests, submitting a patch only if the test it wrote for that issue fails before the patch and passes after it.

AI USE STATEMENT

We used large language models as writing assistants in preparing this paper. We used generative AI tools to write part of the experiment code and part of the code implementing the data-filtering pipeline. Their outputs were checked by execution rather than by inspection. We have reviewed all AI-assisted work and take responsibility for the final content of this paper, including its text, its claims and the artifacts we release.

REFERENCES

Toufique Ahmed, Martin Hirzel, Rangeet Pan, Avraham Shinnar, and Saurabh Sinha. TDD-Bench Verified: Can LLMs generate tests for issues before they get resolved? *arXiv preprint arXiv:2412.02883*, 2024.

Toufique Ahmed, Jatin Ganhotra, Rangeet Pan, Avraham Shinnar, Saurabh Sinha, and Martin Hirzel. Otter: Generating tests from issues to validate SWE patches. *arXiv preprint arXiv:2502.05368*, 2025a.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Heterogeneous prompting and execution feedback for SWE issue test generation and selection. *arXiv preprint arXiv:2508.06365*, 2025b.

Toufique Ahmed, Jatin Ganhotra, Avraham Shinnar, and Martin Hirzel. Reproduction test generation for Java SWE issues. *arXiv preprint arXiv:2605.04320*, 2026.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

Zhi Chen, Zhensu Sun, Yuling Shi, Chao Peng, Xiaodong Gu, David Lo, and Lingxiao Jiang. Rethinking the value of agent-generated tests for LLM-based software engineering agents. *arXiv preprint arXiv:2602.07900*, 2026.

Runxiang Cheng, Michele Tufano, Jürgen Cito, José Cambronero, Pat Rondon, Renyao Wei, Aaron Sun, and Satish Chandra. Agentic bug reproduction for effective automated program repair at Google. *arXiv preprint arXiv:2502.01821*, 2025.

Jeff Da, Clinton Wang, Xiang Deng, Yuntao Ma, Nikhil Barhate, and Sean Hendryx. Agent-RLVR: Training software engineering agents via guidance and environment rewards. *arXiv preprint arXiv:2506.11425*, 2025.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Vijay Bharadwaj, Jeff Holm, Raja Aluri, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

FAIR CodeGen team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, et al. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, et al. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. `https://github.com/harbor-framework/harbor`, 2026.

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. In *Conference on Language Modeling (COLM)*, 2025.

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

Lara Khatib, Noble Saji Mathews, and Meiyappan Nagappan. AssertFlip: Reproducing bugs via inversion of LLM-generated passing tests. *arXiv preprint arXiv:2507.17542*, 2025.

Dongjun Lee, Changho Hwang, and Kimin Lee. Learning to generate unit test via adversarial reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2026.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*, 2026.

Zi Lin, Sheng Shen, Ilia Kulikov, Jingbo Shang, Jason Weston, and Yixin Nie. Learning to solve and verify: A self-play framework for code and test generation. *arXiv preprint arXiv:2502.14948*, 2025.

Zexiong Ma, Chao Peng, Pengfei Gao, Xiangxin Meng, Yanzhen Zou, and Bing Xie. SoRFT: Issue resolving with subtask-oriented reinforced fine-tuning. *arXiv preprint arXiv:2502.20127*, 2025.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. SWT-Bench: Testing and validating real-world bug-fixes with code agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Noor Nashid, Islem Bouzenia, Michael Pradel, and Ali Mesbah. Issue2Test: Generating reproducing test cases from issue reports. *arXiv preprint arXiv:2503.16320*, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. *arXiv preprint arXiv:2412.21139*, 2024.

Xiaoxuan Peng, Kaiqi Zhang, Xinyu Lu, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. LiteCoder-Terminal: Scaling long-horizon terminal environments for learning language agents. *arXiv preprint arXiv:2605.29559*, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL `https://qwen.ai/blog?id=qwen3.5`.

Qwen Team. Qwen3.8-Max, August 2026b. URL `https://qwen.ai/blog?id=qwen3.8-max`.

Shreshth Rajan. Auditing reward hackability in code RL training environments. *arXiv preprint arXiv:2606.16062*, 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, et al. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Qijia Shen, Zhiqi Huang, Vamsidhar Kamanuru, Aznaur Aliev, Jay Rainton, Ahmed Awelkair, et al. SETA: Scaling environments for terminal agents. *arXiv preprint arXiv:2607.10891*, 2026.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, et al. SWE-RM: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*, 2025.

Aditya Bharat Soni, Rajat Ghosh, Vaishnavi Bhargava, Valerie Chen, and Debojyoti Dutta. SWE-Tester: Training open-source LLMs for issue reproduction in real-world repositories. *arXiv preprint arXiv:2601.13713*, 2026.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, et al. BugPilot: Complex bug generation for efficient learning of SWE skills. *arXiv preprint arXiv:2510.19898*, 2025.

Aozhe Wang, Yuchen Yan, Nan Zhou, Zhengxi Lu, Weiming Lu, et al. Code-A1: Adversarial evolving of code LLM and test LLM via reinforcement learning. *arXiv preprint arXiv:2603.15611*, 2026a.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. SWE-Dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics (ACL)*, 2025a.

Junyi Wang, Jialun Cao, and Zhongxin Liu. iCoRe: An iterative correlation-aware retriever for bug reproduction test generation. *arXiv preprint arXiv:2604.19224*, 2026b.

Xinchen Wang, Pengfei Gao, Xiangxin Meng, Chao Peng, Ruida Hu, Yun Lin, and Cuiyun Gao. AEGIS: An agent-based framework for general bug reproduction from issue descriptions. In *ACM International Conference on the Foundations of Software Engineering (FSE)*, 2025b.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, et al. OpenHands: An open platform for AI software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving LLM coder and unit tester via reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025c.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, et al. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

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 (ICML)*, volume 306 of *PMLR*, 2026.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Yingce Xia, Di He, Tao Qin, Liwei Wang, Nenghai Yu, Tie-Yan Liu, and Wei-Ying Ma. Dual learning for machine translation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2016.

Jiacheng Xu, Wentao Zhang, Zhiyi Lyu, Fuxiang Zhang, Chaojie Wang, Yang Liu, and Bo An. Two-stage reinforcement learning for sound and adversarial test generation in code LLMs. *arXiv preprint arXiv:2609.03955*, 2026.

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. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, et al. SWE-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025a.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, et al. Kimi-Dev: Agentless training as skill prior for SWE-agents. *arXiv preprint arXiv:2509.23045*, 2025b.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. SelfAPR: Self-supervised program repair with test execution diagnostics. *arXiv preprint arXiv:2203.12755*, 2022.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025.

Z.ai. GLM-5.3: Frontier coding with emergent cyber capabilities, August 2026. URL https://z.ai/blog/glm-5.3.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, et al. SWE-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

Yiwen Zhang, Xiaodong Yan, Zhenyu Huang, Deng Zhao, Liang Jiang, Qing Cui, Zujie Wen, Zhiqiang Zhang, and Jun Zhou. Robust code RL via faulty-code-driven test case synthesis and dense reward shaping. *arXiv preprint arXiv:2608.24135*, 2026.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, et al. Immersion in the GitHub universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*, 2026.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments. *arXiv preprint arXiv:2512.12216*, 2025.

A Keeping the Answer out of the Agent's Reach

Both tasks are defined by withholding one of the instance's artifacts, so the reward means nothing if the agent can read the artifact it is being asked to reconstruct. A submitted test, moreover, runs with the same privileges as the verifier that grades it. Every measure below is therefore enforced when the task is built or after the agent's session has ended, rather than by constraining what the agent's own process may do.

The graded artifacts are not in the image. The image the agent works in is built from the buggy snapshot with the reference tests deleted. The reference tests and the reference patch are kept outside that image and are mounted into the container only at scoring time, once the session has ended.

The repository cannot be used to recover them. Repository history is scrubbed when the task is built, so the commit that resolves the issue is not reachable from inside the container. The container has no network access, so the upstream repository, the pull request and the published package cannot be fetched either.

The submission path is fixed in advance. The verifier knows which file to execute before the agent starts, and test files that existed before the session are restored to their pre-agent state before grading. An agent therefore cannot collect the reward by editing a test that already passes, and cannot hide a submission the verifier will not look at.

B Conversion Funnel Details

This appendix gives the counts behind Table 1.

Scale-SWE. Conversion keeps 8,243 of the 17,470 instances that reproduce. The conditions of Section 3 are checked from the task files alone, without running a model or a container. They are applied in the order below, and each count is the number of instances a condition removes after the earlier ones.

- The reference patch leaves the agent's test file, the root `conftest.py` and the pytest configuration untouched: 85.
- The problem description is at least 614 characters long, which is the first percentile of this source (the median is 1,365): 173.
- The verifier has at least one hunk of the reference patch to revert, counting only hunks in library code that change more than text: 1,281.
- The instance concerns a single behaviour: at most four such hunks, fewer than two numbered subsections in the problem description, and no hunk whose change has no runtime effect: 7,688.

The limit of four hunks is the loosest one at which a hand-labelled set kept no instance that bundles unrelated changes. Raising it to six would keep 1,205 more instances. Of the 8,243 converted tasks, 1,287 were not admitted.

SWE-smith-py. Before the reference tests are run, we remove the 9,564 instances made by the generators that combine several injected edits (`combine_file`, `combine_module`) or rewrite a whole function (`lm_rewrite`). Of the remaining 22,309 instances, 21,441 reproduce. We then convert only the 14,390 whose resolution task was admitted. The other 7,051 were not admitted, mostly because a damaged version of the reference patch still passed the reference tests (4,768) or because no damaged version could be generated (1,909). Conversion rejects a further 38 bundled instances, leaving 14,352. Of these, 1,102 were not admitted: 226 never received a verdict from the verifier, and 876 received one that did not admit them.

R2E-Gym. Conversion removes 39 of the 8,049 instances that reproduce: 35 whose reference patch touches no library source, and 4 excluded when the list conversion runs on was assembled. For 2,317 of the 8,010 converted tasks no damaged version of the reference patch is available. In 2,169 of them no damaged version makes a fail-to-pass reference test fail, in 121 the fail-to-pass tests cannot be identified unambiguously by name, and in 27 the damaged versions could not be reproduced. Of the remaining 5,693 tasks, 3,314 are admitted. A permissive count that also accepts a failure by an exception raised in the code under test, rather than only a failing assertion, admits 3,794.

C Admission Checks

Admission has two parts. Every converted task is run once with its own fail-to-pass reference tests in place of the agent's submission, as Section 4.2 describes. Separately, the verifier is run against tests written to take each shortcut of Section 3.3, which it must refuse.

When a task is run with its reference tests before training, a damaged version counts as caught only when a test fails on it through a failing assertion. The verifier also records which lines execute beyond those an import alone runs, and whether they reach the statements the reference patch modified; line numbers past the end of each file are mixed into the lines it asks about, so that a record copied rather than produced by execution is visible. A task on which the reference tests leave no such record is not admitted.

The honest probe. The honest probe is the instance's own fail-to-pass reference tests rather than a test we wrote, so what it establishes is that the verifier rewards the tests the instance was built around.

Shortcut probes. Table 6 gives the six families of shortcut the verifier was probed with on Scale-SWE tasks, and how many runs of each it refused. A run is one probe on one task, and it counts as refused when the submission receives no reward. The denominator counts runs for which a verdict was recorded rather than verifiers containing the probe, since the probe code is compiled into every verifier and counting verifiers would overstate the sample by two orders of magnitude. It excludes one task on which no damaged version of the reference patch is available, which is a task admission removes, since there the trial against damaged fixes never runs.

What the agent actually submitted. Probes establish what the verifier does when a shortcut is handed to it. Whether shortcuts arise on their own, and whether they are rewarded, is a question about the submissions collected during training. We recovered the text of 1,790 test-generation submissions from the two reinforcement learning runs and classified each of them by static inspection, using patterns written to miss a borderline case rather than to count an honest test as a shortcut. A submission counts as comparing source text only if it both reads a `.py` file and tests the content it read for a substring; as fingerprinting structure only if it reads source and also calls a parser or a hash; and as reading earlier state only if it reads an environment variable or a file outside the repository, so that writing a temporary fixture does not qualify.

Every rewarded submission that matches a shape was read, 26 submissions over the 27 entries of the last column since one of them matches two shapes, and in each one the shape is a property of the

Table 6: Shortcut probes, by family. Each of the 86 runs writes a test that takes the shortcut named in the row, submits it to the verifier of one Scale-SWE task, and is refused when the verifier gives it no reward.

Shortcut the probe takes	Runs	Refused
Asserts nothing	16	16
Checks that the buggy source text is absent	15	15
Checks that the fixed source text is present	16	16
Fingerprints the syntax tree	13	13
Fingerprints the recomputed normalisation	13	13
Reads state left by an earlier trial	13	13
All six families	86	86

Table 7: Test-generation submissions collected during training, classified by static inspection. A row counts the submissions in whose text the shape appears. *Rewarded* counts those that received the reward, in full or with a deduction for not following the submission protocol. Two submissions match two shapes, so the rows sum to 1,792.

Shape of the submitted test	Submissions	Share	Rewarded	Rewarded with no behavioural content
Asserts nothing	23	1.3%	8 (35%)	0
Compares the source text	12	0.7%	6 (50%)	0
Fingerprints the structure of a file	0	0.0%	—	—
Reads state left by an earlier trial	29	1.6%	13 (45%)	0
Reads or writes the verifier	0	0.0%	—	—
None of these	1,728	96.5%	1,153 (67%)	—

pattern rather than an absence of behaviour. A test of a code formatter asserts the text the formatter outputs; a configuration library takes environment variables as its input, so a test that sets one is feeding it; a test that reproduces a crash needs no assertion of its own, since the exception raised by the code under test fails it; and the pattern for reading source matches a test that writes a `.py` file as a fixture as well as one that reads the library's own. No shape is rewarded as often as a submission matching none of them. The submissions counted here are those that entered a gradient update, which is the population whose text was retained; the sampled trajectories that a zero-variance group removed before the update were not kept and cannot be classified.

D Conversion on Terminal Tasks

The conversion does not depend on the task being a repository issue. A terminal task carries four artifacts of the same kind, and exchanging which of them is withheld produces a second task in the same way. This appendix records the collection we applied it to, how a task is converted, and how the resulting tasks were trained on.

Source. LiteCoder-Terminal-RL (Peng et al., 2026) supplies 602 terminal tasks. Each holds an instruction, the container image the agent starts in, a reference solution written as a shell script, and a checker that decides whether the instruction was carried out. We chose it among five candidate collections by profiling each of them against Terminal-Bench 2.1 with one script: it is the collection that matches the benchmark most closely on network access, on the processor and memory limits its tasks request, and on how their checkers decide an outcome.

Conversion. Solving a terminal task withholds the reference solution and judges the agent's work with the checker. Under the opposite division the agent receives the instruction with a preface telling it not to carry the instruction out, but to write a script that decides whether it was carried out, at a path fixed in advance; the reference solution and the original checker are withheld. The verifier of the converted task establishes by execution what a correct outcome is: it records the state of the

working directory, runs the reference solution, records the state again, and takes the difference. It then constructs outcomes that are wrong by mutating the correct one, and keeps a mutation only when the task's own checker rejects it, which is the counterpart of the damaged reference patches of Equation 4. The reference solution and the original checker are deleted before the submitted script is executed. The script is rewarded, on the same binary scale and with the same deductions for not following the submission protocol as in Section 5, only when it accepts the correct outcome and rejects every wrong one.

Which tasks convert. Of the 602 tasks, 63 have a checker that queries a running service rather than the state of the file system, and the converted task cannot ask for a script that decides those. The condition that binds first, however, is that the task be executable as given: a task is kept only when its own reference solution, run in its own image, actually earns the reward from its own checker, which 300 of the 602 do over three rounds of execution. All 300 convert, and 230 of them are admitted, admission requiring that both the original task and the converted one be usable.

Training. The converted tasks were trained on in three runs of 20 steps, which differ only in the tasks they draw: one on the terminal tasks alone, 250 of them; one on the converted tasks alone, 190; and one on both, 440. The held-out tasks are split by the identity of the source task, so that the two directions of one task never fall on opposite sides. The remaining settings follow Section 5: GRPO with a group of 8, 4 tasks entering each update out of 12 sampled, 24 for the run on converted tasks alone, a global batch of 32, a constant learning rate of 10^{-6}, no KL penalty, a clipping range of 0.2, and groups without reward variance discarded. The agent works within 80 turns, a context of 65,536 tokens and a response of 16,384.

Table 8: Tasks solved on Terminal-Bench 2.1, of 89. All four models start from Qwen3.5-9B, and the three trained ones are the 20-step runs described above, which differ only in the tasks they draw.

		Trained on one task		
	Base	Terminal	Converted	Joint
Tasks solved	18	18	18	21
– of 89	20.2%	20.2%	20.2%	23.6%

Results. Table 8 reports the four models on all 89 tasks of Terminal-Bench 2.1 (Merrill et al., 2026), evaluated on one infrastructure and with one set of inference settings. Training on the terminal tasks alone and training on the converted tasks alone each leave the count where the base model leaves it, and training on both solves three tasks more. Three tasks out of 89 is a marginal difference, so we leave a larger training budget and a repeated evaluation protocol to future work.