Cross-Benchmark Transfer from RL on Agentic Coding Tasks

Surge AI Research
research@surgehq.ai

Abstract

Coding agents can have several failure modes: they can drop subtle requirements, test only the paths their implementation already handles, break working behavior that was supposed to stay intact, or validate against false or invalid assumptions. We investigate whether reinforcement learning on expert-built agentic coding tasks can help close this last-mile gap, and whether what the agent learns also generalizes beyond the training distribution. We post-train Kimi K2.7 with RL on 1,700 expert-built tasks: 1,000 repository tasks with hidden fail-to-pass and pass-to-pass tests, and 700 Terminal-Bench-style tasks with expert-written hidden verifiers. The reward is dense (based on the fraction of target criteria satisfied), and a rollout that fails any existing pass-to-pass test receives zero reward. Trained with GSPO and a rank-32 LoRA adapter for a single epoch, the model improves on all five external benchmarks we evaluated: SWE-Bench Pro ($60.1 \rightarrow 64.8$, $+4.7$ pp), DeepSWE ($31.0 \rightarrow 43.4$, $+12.4$ pp), Terminal-Bench 2.1 ($67.4 \rightarrow 82.0$, $+14.6$ pp), Terminal-Bench 3 ($1.4 \rightarrow 12.1$, $+10.7$ pp), and SWE-Marathon ($5.0 \rightarrow 25.0$, $+20.0$ pp). Median trajectory length fell by 24–35% in agent steps. A SxS paired analysis of base and trained trajectories shows that the base model's DeepSWE failures are typically near-misses and identifies recurring failure modes: missed requirements, narrow testing, and silent regressions, each of which the trained model avoided on the same tasks.

1 Introduction

Reinforcement learning with verifiable rewards has become the standard tool for improving language-model coding agents [DeepSeek-AI et al., 2025, Wei et al., 2025, Luo et al., 2025, Kimi Team et al., 2025]. The standard evidence that this approach works is an increase in corresponding benchmark scores; however, this leaves two questions open. First, what did the model learn: specific domain knowledge, or something more fundamental about carrying a task through to completion? Second, does the improvement survive a change of benchmark or agent harness, on tasks significantly different from the training data? These questions matter beyond any specific leaderboard: an agent that has learned a benchmark's nuances and conventions is worth less than one that has learned to finish a realistic task.

We look into both questions with a single training run and multiple evaluation benchmarks. We post-train Kimi K2.7 with reinforcement learning alone on 1,700 expert-built agentic coding tasks drawn from two families (§3). The 1,000 repository tasks use a pinned commit and require a change verified by hidden fail-to-pass tests, with pass-to-pass tests guarding existing behavior. The 700 terminal tasks use a working environment and require a deliverable tested by expert-written hidden verifiers. We use a straightforward recipe so that the data, not the algorithm, is what is tested: GSPO [Zheng et al., 2025] on a rank-32 LoRA adapter [Hu et al., 2022], no supervised warm-up, and a reward that depends on the number of satisfied checks and, on repository tasks, drops to zero whenever a pass-to-pass test regresses (§4).

Preprint.

We then evaluate the base and trained models on five external benchmarks, each through one fixed agent harness (§5). Two of them, SWE-Bench Pro [Deng et al., 2025] and Terminal-Bench 2.1 [Merrill et al., 2026, Terminal-Bench Team, 2026], share the task formats of the training data; the other three differ in various ways: DeepSWE [Huang et al., 2026] requires from-scratch, multi-file feature work in existing open-source repositories; Terminal-Bench 3 [Marten et al., 2026] extends terminal work into hardware, scientific computing, machine learning, security, and operations; and SWE-Marathon [Desai et al., 2026] asks agents to build complete systems over multi-hour trajectories, a horizon and scope with no counterpart in training. The training set was checked against all benchmarks and shares no tasks or data with them. The five are run through three harnesses with different tool interfaces, prompts, and control loops.

The trained model scores higher on all five benchmarks (§6; Figure 1): by $+4.7$ pp on SWE-Bench Pro, $+12.4$ pp on DeepSWE, $+14.6$ pp on Terminal-Bench 2.1, $+10.7$ pp on Terminal-Bench 3, and $+20.0$ pp on SWE-Marathon. The trained model also used fewer agent steps where we measured them: the median fell from 150 to 98 on DeepSWE and from 102 to 78 on Terminal-Bench 3.

To better understand what changed, we paired base and trained trajectories on the same tasks and analyzed the models' reasoning and outputs (§7). On DeepSWE, the base model's failures are typically near-misses: among its failed runs, the median run still passed 86% of the target tests, and 84% preserved every existing pass-to-pass test. The analyzed failures fall into four main buckets: missed requirements (most of the feature is built, but some constraints were dropped), narrow testing (tests were designed around the cases the implementation already handled), silent regressions (new objectives satisfied at the cost of existing behavior), and weak verification against unchecked and faulty assumptions. In the corresponding trajectories from the trained model, the model implemented the full specification, derived its tests from the requirement instead of its own implementation, and better tracked what behavior must keep working.

Contributions:

1. Post-training on expert-built coding tasks that improves five external benchmarks, with fewer agent steps.
2. A side-by-side trajectory analysis of the base model's failures and a taxonomy of these failure modes, each paired with the trained model's behavior on the same task in a concrete before/after case.
3. Reward design (dense criterion credit gated on regressions) and the training configuration (§4).

2 Related Work

Reinforcement learning for software-engineering agents. Reinforcement learning with verifiable rewards (RLVR) was established on mathematics [Shao et al., 2024, Lambert et al., 2024] and, with DeepSeek-R1, on short-form code [DeepSeek-AI et al., 2025], and has since moved to long-horizon agentic software engineering. SWE-RL [Wei et al., 2025] learns from software-evolution data with a rule-based patch-similarity reward; SWE-Gym [Pan et al., 2025], SWE-smith [Yang et al., 2025], and R2E-Gym [Jain et al., 2025] build executable training environments from real repositories at scale; DeepSWE-Preview [Luo et al., 2025] trains an agent with RL alone in such environments, and SkyRL [Cao et al., 2025] provides an RL pipeline for long-horizon agents; frontier open models such as Kimi K2 [Kimi Team et al., 2025] rely on large-scale agentic data synthesis with RL. These efforts largely evaluate on SWE-bench-style benchmarks [Jimenez et al., 2024, Chowdhury et al., 2024] whose format matches the training environments. Our training tasks are expert-authored with hidden verifiers and explicit regression gates, and we evaluate on multiple benchmarks to measure transfer rather than in-distribution improvement.

Policy optimization. We use GSPO [Zheng et al., 2025], which replaces GRPO's [Shao et al., 2024] token-level importance ratios with length-normalized sequence-level ratios; truncated importance sampling to correct for the numerical mismatch between the inference engine used for rollouts and the training engine [Yao et al., 2025]; overlong filtering of truncated rollouts [Yu et al., 2025]; and a low-rank adapter [Hu et al., 2022]. Schulman and Thinking Machines Lab [2025] report that LoRA matches full fine-tuning for RL post-training, whose per-episode information content is small, at a fraction of the memory cost.

Benchmarks for agentic coding. SWE-bench [Jimenez et al., 2024] and its verified subset [Chowdhury et al., 2024] established issue resolution on real repositories as the standard task; SWE-Bench Pro [Deng et al., 2025] raises difficulty with long-horizon, multi-file, human-verified problems. Terminal-Bench [Merrill et al., 2026] evaluates agents on hard tasks in containerized command-line environments and, through the Harbor framework [Harbor Framework Team, 2026], has become a continuously versioned benchmark (2.0, 2.1, 3.0, 4.0) [Terminal-Bench Team, 2026, Marten et al., 2026, Marten, 2026]. Two 2026 benchmarks, DeepSWE [Huang et al., 2026] and SWE-Marathon [Desai et al., 2026], require building from-scratch features and multi-hour builds (§5). Scores on all of these depend substantially on the agent scaffold [Merrill et al., 2026, Desai et al., 2026] (SWE-agent and mini-swe-agent, OpenHands, Terminus 2, and commercial CLIs such as Claude Code [Yang et al., 2024, Wang et al., 2025, Harbor Framework Team, 2025, Anthropic, 2025]), which controls the tools, prompts, and loop the model operates in. We therefore fix one harness per benchmark and use it unchanged for both models (§5).

Dense and rubric-based rewards. Partial-credit rewards derived from expert-written criteria have been used to train instruction following [He et al., 2025, Mehta et al., 2026a] and open-ended reasoning [Gunjal et al., 2025]. Our reward is an analogue for coding: the fraction of hidden target checks that pass. Proxy rewards can be gamed [Skalse et al., 2022], and coding agents exhibit reward hacking even under binary verification; SWE-Marathon reports it in 13.8% of frontier-agent rollouts [Desai et al., 2026]. Our reward addresses one specific exploit of partial credit: satisfying new checks at the expense of existing behavior, by assigning a zero reward whenever a pass-to-pass test fails (§4.1).

Transfer and generalization from post-training. Chu et al. [2025] find that RL generalizes to unseen rule and visual variants where supervised fine-tuning memorizes; Huan et al. [2025] find that RL on mathematics transfers to other domains better than supervised fine-tuning on the same problems; and Yue et al. [2025] argue that much of RLVR's pass@1 gain on reasoning tasks reflects concentrating probability on solutions the base model could already sample, so that pass@k at large k does not improve. Prior work from our group reports transfer across task distributions: training on long-horizon office work with no coding tasks improved SWE-Bench Pro by $+5.8$ pp [Ritchie et al., 2026], multi-tool post-training produced consistent gains across five external evaluations [Mehta et al., 2026c], single-turn constraint-following training produced its largest gains on the multi-turn axes of MultiChallenge [Mehta et al., 2026a, Sirdeshmukh et al., 2025], and training on an enterprise simulation improved out-of-distribution pass rates [Mehta et al., 2026b]. The training and evaluation domains in this experiment are both coding, but the benchmarks and scaffolds differ considerably.

3 Training Data

The training set consists of 1,700 tasks drawn from Surge AI's expert-built agentic coding task collections, split across two task families, using the formats common in agentic coding. Every task ships with a specification the agent sees and a set of hidden checks it does not.

3.1 Two task families

Repository tasks (1,000). Each task consists of a real repository at a pinned commit and requires the agent to make a specific realistic change. Two groups of hidden tests are used to grade the result. Fail-to-pass tests encode the requested behavior and fail on the original commit; pass-to-pass tests guard existing behavior that must stay as is. The format is that of SWE-bench [Jimenez et al., 2024], but the tasks are created from scratch and the specifications are concrete, individually gradable requirements, including edge cases, exact interfaces, output formats, restrictions, and constraints on what must remain unchanged.

Terminal tasks (700). Each task places the agent in a working containerized environment and asks it to produce a concrete deliverable, in the style of Terminal-Bench [Merrill et al., 2026]. Expert-written hidden verifiers inspect the final state of the environment and score a set of criteria.

All training tasks are checked for contamination against common evaluation benchmarks [Deng et al., 2025, Huang et al., 2026, Terminal-Bench Team, 2026, Marten et al., 2026, Desai et al., 2026].

3.2 Reward design

Requirements are explicit and individually gradable, so the reward is a function of the number of criteria passed (§4.1), and a rollout that satisfies more criteria ranks above one that satisfies fewer, even when neither rollout passes every check. The grader is always hidden, so the model sees the required specification but never the tests or verifier that determine its final reward. This implies that the model cannot code against visible test cases and has to determine for itself which behaviors are worth checking. Existing correct behavior is also protected by specifications that state what must not change. Satisfying every new requirement while breaking one existing test still gets a score of zero.

4 Method

The training recipe is a single-epoch RL run on the 1,700 tasks. The tasks were used in a coarse two-stage curriculum, the 1,000 repository tasks first and the 700 terminal tasks after them. The order is based on difficulty: the base model's pass rate was slightly higher on the repository tasks, and the terminal tasks were harder. Within each stage the prompts were shuffled.

4.1 Reward

Each rollout ends with the hidden checks being executed against the final environment state. Let $P_1, \ldots, P_{N_P} \in \{0, 1\}$ be the outcomes of the PASS_TO_PASS tests (existing behavior) and $F_1, \ldots, F_{N_F} \in [0, 1]$ the outcomes of the FAIL_TO_PASS tests or verifier criteria (requested behavior), all criteria weighted equally. The reward is

$$R \;=\; \mathbf{1}\left[\sum_{i=1}^{N_P} P_i = N_P\right] \cdot \frac{1}{N_F} \sum_{j=1}^{N_F} F_j. \tag{1}$$

The second component gives dense feedback: a rollout that satisfies 3 of 12 requested checks scores 0.25; one that satisfies 7 of 10 scores 0.7. The first factor is also a hard requirement: a rollout that satisfies all 12 requested checks but fails a single one of 100 pass-to-pass tests gets a score of 0. Terminal tasks do not have pass-to-pass tests ($N_P = 0$), so the reward is the fraction of verifier criteria satisfied. Rollouts that hit the response-length cap are dropped rather than scored, as in DAPO's overlong filtering [Yu et al., 2025]; dropped rollouts also shrink the group used for the advantage baseline.

With group-relative advantages (§4.2) the reward enters the update only through the standardized advantages within a group of rollouts for the same prompt, so the dense partial scoring does two things. In a group where no rollout passes every check (which would have received a score of 0 under a binary reward scheme), it spreads the advantages based on how many of the criteria each rollout satisfied. The regression check then assigns zero reward to any rollout that regresses existing behavior regardless of how many new requirements it satisfies, so there is no exchange rate between new and existing behavior for the policy to exploit. Both properties reappear as behaviors we further discuss in §7.

4.2 Policy optimization

We optimize the policy with Group Sequence Policy Optimization (GSPO; Zheng et al., 2025), whose importance ratio for a sampled response y is the length-normalized sequence ratio $s(\theta) = \left(\pi_\theta(y \mid x)/\pi_{\theta_{\text{old}}}(y \mid x)\right)^{1/|y|}$. GSPO was designed for stable RL on mixture-of-experts models, and its sequence-level ratio better tolerates the rollout–trainer numerical mismatch [Zheng et al., 2025]. For each prompt we sample a group of $G = 8$ rollouts and compute group-relative advantages. Each rollout step samples 16 prompts, for 128 trajectories, consumed in two optimizer updates of 64 sequences each. We over-sample prompts (in batches of 16, with up to eight retries) and discard groups whose rewards have no variance, so that every step is filled with groups that carry some learning signal, in the spirit of dynamic sampling [Yu et al., 2025]. Rollouts use a 65,536-token per-turn context, nucleus sampling with $p = 0.95$, and the model's thinking mode.

Table 1: The five external benchmarks [Deng et al., 2025, Huang et al., 2026, Merrill et al., 2026, Terminal-Bench Team, 2026, Marten et al., 2026, Desai et al., 2026]. Repository and terminal task formats are present in training (§3.1); SWE-Marathon's multi-hour horizon and scope are not. One scaffold was fixed per benchmark and used unchanged for both models (§5).

Benchmark	Tasks	What it asks for	Scaffold
SWE-Bench Pro (public split)	731	Long-horizon issue resolution in 11 real repositories; multi-file, human-verified patches. *Repository format.*	mini-swe-agent
DeepSWE v1.1	113	From-scratch features across 91 existing open-source repositories in five languages; behavior-focused tests run in a separate verifier container; reference solutions average about 670 added lines. *Repository format.*	mini-swe-agent
Terminal-Bench 2.1	89	Hard tasks in containerized command-line environments spanning software engineering, system administration, data processing, model training, and security [Artificial Analysis, 2026]. *Terminal format.*	Terminus 2
Terminal-Bench 3	74	Harder, longer-running terminal tasks across seven domains (software, science, ML, operations, security, hardware, media); open-internet. *Terminal format.*	mini-swe-agent
SWE-Marathon v1.1	20	Build complete systems (library and product clones, ML engineering, algorithmic work) in containerized environments under 2–10-hour agent time limits; expert estimates 40–400 hours. *Terminal format; multi-hour horizon not in training.*	Claude Code

Low-rank adaptation. Rather than updating all parameters of the MoE, we train a rank-32 LoRA adapter [Hu et al., 2022] with $\alpha = 32$. The adapter is trained with Adam and decoupled weight decay [Kingma and Ba, 2015, Loshchilov and Hutter, 2019].

4.3 Infrastructure and training run

Training ran on NVIDIA H200 GPUs with rollout generation and policy updates co-located on the same devices. The stack is built from open-source components: Miles [Miles Team, 2026], an RL framework derived from slime [Zhu et al., 2025], orchestrates SGLang [Zheng et al., 2024] for rollout generation and Megatron-LM with Megatron-Bridge [Shoeybi et al., 2019, NVIDIA, 2025] and FlashAttention [Dao et al., 2022] for training, combining tensor, pipeline, context, and expert parallelism to fit the MoE. All rollouts execute inside an isolated sandbox that provides the task's repository or environment and runs the hidden checks at the end of the rollout.

All results use the final checkpoint, fixed in advance, and no hyperparameter was tuned for any benchmark.

5 Evaluation Protocol

Benchmarks. Table 1 summarizes the five benchmarks. SWE-Bench Pro [Deng et al., 2025] and Terminal-Bench 2.1 [Terminal-Bench Team, 2026] share the task structure and format with the training data. DeepSWE [Huang et al., 2026] consists of tasks written from scratch, not adapted from existing commits. Terminal-Bench 3 [Marten et al., 2026] extends Terminal-Bench 2.1 with longer-running tasks across seven domains; and SWE-Marathon [Desai et al., 2026] requires work that experts estimate at 40–400 hours, in containerized environments with hidden checks like our terminal tasks but at a horizon and scope with no counterpart in training. These five are the only benchmarks we evaluated, and each in-house score is from a single, first run.

Harnesses. Scores can vary substantially across harnesses [Merrill et al., 2026, Desai et al., 2026], so we fixed one per benchmark and used it unchanged for the baseline and trained model checkpoints: mini-swe-agent [Lieret and Jimenez, 2025, Yang et al., 2024] for SWE-Bench Pro (accepted on the public leaderboard, whose default scaffold is SWE-agent), for DeepSWE (the leaderboard's scaffold for all models), and for Terminal-Bench 3 (Terminal-Bench designates no default; its reference agent is Terminus 2); Terminus 2 [Harbor Framework Team, 2025, 2026] for Terminal-Bench 2.1; and Claude Code [Anthropic, 2025] for SWE-Marathon, one of the scaffolds evaluated in the benchmark paper, connected to the model through an API-compatible endpoint.

Table 2: Pass@1 (%) of Kimi K2.7 before and after RL on 1,700 expert-built coding tasks. The baseline marked † is a publicly reported score for the same checkpoint and scaffold (§5).

Benchmark	n	Scaffold	Base	+RL	Gain (pp)
SWE-Bench Pro	731	mini-swe-agent	60.1	**64.8**	+4.7
DeepSWE	113	mini-swe-agent	31.0	**43.4**	+12.4
Terminal-Bench 2.1	89	Terminus 2	67.4†	**82.0**	+14.6
Terminal-Bench 3	74	mini-swe-agent	1.4	**12.1**	+10.7
SWE-Marathon	20	Claude Code	5.0	**25.0**	+20.0

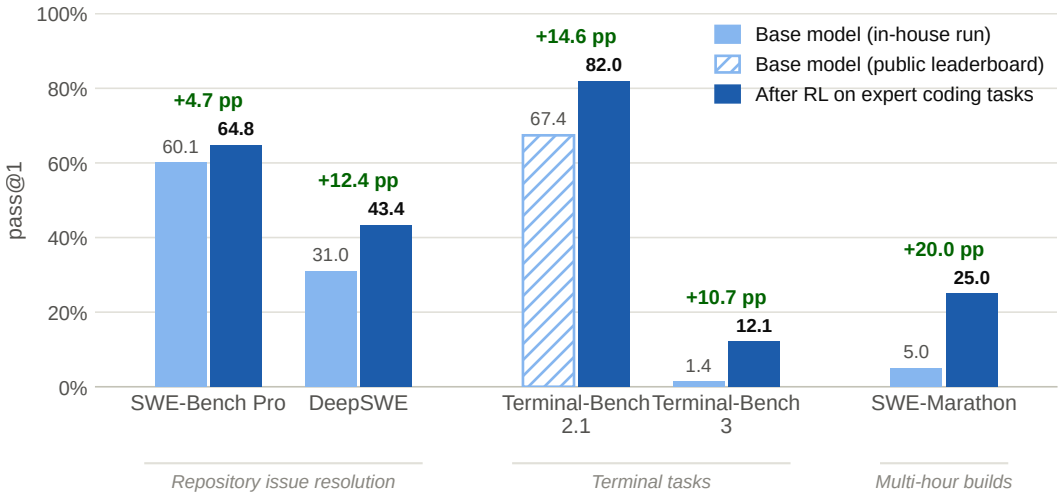

Figure 1: Pass@1 on five external coding benchmarks before and after RL on 1,700 expert-built coding tasks, grouped by the two task formats present in training, with SWE-Marathon's multi-hour builds shown separately. The hatched baseline (Terminal-Bench 2.1) is a publicly reported score, not an in-house run.

Metric and settings. All in-house results are pass@1 from a single run per benchmark using the final checkpoint, in the model's thinking mode; the Terminal-Bench 2.1 baseline averages three runs (below). Both models were run through identical harnesses.

Baseline runs. For Terminal-Bench 2.1 we use the score reported by Artificial Analysis [Artificial Analysis, 2026], which runs the Terminus 2 scaffold in its own sandbox and averages pass@1 over three repeats. It evaluates the same public checkpoint under the same scaffold we used for the trained model.

6 Results

6.1 The trained model scores higher on all five benchmarks

The trained model scores above the base model on every benchmark (Table 2, Figure 1). The gains correspond to roughly 35, 14, 13, 7–8, and 4 additional solved tasks on SWE-Bench Pro, DeepSWE, Terminal-Bench 2.1, Terminal-Bench 3, and SWE-Marathon, respectively. Two of the five gains are also individually significant at the 0.05 level (Terminal-Bench 2.1 and Terminal-Bench 3).

6.2 Where the gains appear

Improvements appear on all task shapes and formats, on repository-style (+4.7 and +12.4 pp) and terminal-style tasks (+14.6 and +10.7 pp), and on SWE-Marathon's multi-hour builds (+20.0 pp), whose horizon and scope have no representation in training. The improvement is not confined to the two benchmarks whose task sets the training data most closely resembles.

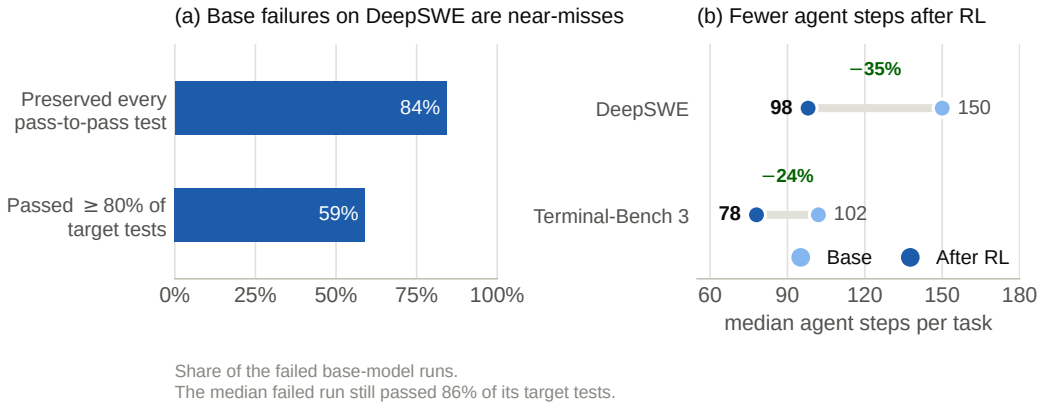

Figure 2: (a) Base-model failures on DeepSWE are often near-misses: of the failed base runs, 84% preserved every pass-to-pass test and 59% passed at least 80% of the target tests. **(b)** The median trajectory length over all tasks (passed or failed) fell from 150 to 98 agent steps on DeepSWE and from 102 to 78 on Terminal-Bench 3 (§6.3).

The training data shares no tasks with any of the five benchmarks, so none of the gains comes from training on benchmark tasks. The gain appears under all three harnesses (§5); since three of the five benchmarks share mini-swe-agent, however, the evidence that the gain does not depend on the harness rests on Terminal-Bench 2.1 (Terminus 2, with a leaderboard baseline) and SWE-Marathon (Claude Code, 20 tasks).

6.3 Fewer agent steps

We also observed shorter agent trajectories (Figure 2b). On paired DeepSWE runs the median trajectory length over all 113 tasks fell from 150 steps to 98 (-35%); on Terminal-Bench 3 it fell from 102 to 78 (-24%).

7 What Changed: A Paired Trajectory Analysis

To investigate changes in model behavior, we paired base and trained rollouts on the same tasks and read the models' reasoning, shell commands, tests, and outputs, concentrating on tasks the base model failed and the trained model passed. The analysis conditions on fail→pass pairs.

7.1 The base model's failures on DeepSWE are near-misses

Kimi K2.7 is already a capable agentic coding model, and its failures reflect that. Among the DeepSWE tasks the in-house base-model rollouts failed, the median failed run still passed 86% of the task's target tests; 59% passed at least 80% of them; and 84% preserved every existing pass-to-pass test (Figure 2a). The typical failure is often almost right: most of the feature built and one or two things missing at the edges. Passing 86% of tests is not satisfying 86% of requirements (in the first case below, one dropped requirement fails 16 of 137 tests); the statistic measures how close failed runs came, not how many requirements they missed.

7.2 Four failure modes and the behaviors observed after RL

In our SxS analysis of paired trajectories, the base model's misses fall into four main patterns, and in each case the trained model's trajectory on the same task shows a different behavior (Table 3). These dimensions map onto a standard software engineering loop: understand the requirement, test it, maintain what already works, verify against correct assumptions. Below we share some representative examples drawn from DeepSWE, SWE-Marathon, and Terminal-Bench 3.

Table 3: Main failure modes of the base model in the SxS paired trajectories, and the trained model's behavior on the same tasks.

Stage	Failure mode before RL	Behavior observed after RL
Understand	**Lost requirements.** Built most of the feature but dropped a required constraint, often some subtle detail.	**Implement the entire specification.** Carries the stated requirements through their complete implementation and final reviews.
Test	**Narrow testing.** Tested the cases its own incomplete implementation was built around.	**Test the requirement, not the current implementation.** Constructs positive, negative, and alternate-form cases from the specification when needed.
Maintain	**Silent regressions.** Satisfied the new requirements while breaking existing behavior that was supposed to remain intact.	**Protect existing behavior.** Tracks both behaviors: what must change and what must continue to work before editing.
Verify	**Weak ground truth.** Validated against false assumptions or a weak proxy.	**Reconstruct ground truth.** Builds an independent reference, emulator, or manufactured test inputs for more rigorous validation.

7.2.1 Lost requirements → implement the entire specification

Before. A FastAPI task listed twenty requirements. Requirement 18 asked a tracking middleware to expose the `get_stats()` and `reset_stats()` methods. The base model implemented almost the entire feature with the right architecture and working code, but placed the two functions at the module level rather than as methods on the middleware object. It passed 121 of 137 target tests; the remaining sixteen called the methods exactly where the specification said they should exist, and all sixteen tests failed.

After. The trained model made them instance methods and passed 137 of 137 tests. Nothing about the base model's understanding of the feature was incorrect. One interface requirement was lost between reading the specification and finishing the implementation, but the trained model also carried this last constraint through.

Counterpart in the training signal. Reward accumulates one criterion at a time (Equation 1), so within a group of rollouts an attempt that satisfies the complete task always ranks above one that satisfies the main gist; dropping even one requirement in twenty affects the final rollout reward.

7.2.2 Narrow testing → test the requirement, not the implementation

Before. Both models test frequently; the difference is in the behavior they choose to test. One task required default parameter values to work for both named and anonymous function definitions. The base model chose an implementation that handled the anonymous form,

```
func(a = 1),
```

and every test it wrote used the same form. The specification also required the named form,

```
func f(a = 1),
```

whose identifier between `func` and the parenthesis the baseline implementation did not handle, and both hidden test suites failed. The model had designed its tests around the path its incomplete implementation already supported.

After. The trained model began with the named example from the specification itself, then tried anonymous functions, multi-line parameter lists, spread calls, immediately invoked functions, and several other variants. Instead of asking whether its implementation worked on the cases it already supported, it looked for corner cases that could prove its interpretation of the requirement wrong.

Counterpart in the training signal. Since the grader is always hidden, the model sees the specification but never the tests that determine its reward, so it cannot code against visible cases and has to decide for itself which behaviors are worth checking, which means testing what was actually asked for and not only what was already built.

7.2.3 Silent regressions → protect existing behavior

Before. A task required the model to eliminate layout shift across a site's pages while preserving the same visible elements, styling, and analytics side effects. The base model achieved zero layout shift on every page, partly by deleting two components. Its own final review judged the deletion as harmless because some of the components' styling had been inlined elsewhere.

After. The trained model noted what the specification asked to remain before it began editing, then preserved these components while changing how their visual effects were applied. Instead of optimizing only for the requested change, it maintained two lists: what needed to change and what was not allowed to.

Counterpart in the training signal. Preservation is baked into the reward design: a rollout that breaks required existing behavior earns zero no matter how many new requirements it satisfies, and ranks last in its group. There is no exchange rate between new and old behavior for the policy to exploit.

7.2.4 Weak ground truth → reconstruct ground truth from the specification

Before. Some tasks provided no reference implementation or method against which to check work. In those cases the base model would sometimes make an assumption and then construct a test that inherited this incorrect assumption. On a two-dimensional spectral-solver task the official reference functions were unavailable inside the container. The base model decided the problem was effectively one-dimensional, built a one-dimensional solver, and validated it against reference data constructed under the same assumption. The test had no way to expose the original faulty assumption.

After. The trained model chose a two-dimensional test solution whose answer it could derive analytically, constructed an input for which that solution had to be correct, and used the pair as an independent check. The check exposed an instability before submission. On a separate Verilog console task, the trained model wrote a Python emulator and compared the hardware implementation against it instruction by instruction. On a SWE-Marathon task that required a Zstandard decompressor in an environment with no reference `zstd` binary, the trained model wrote a compressor first, so that it could manufacture valid compressed inputs and check whether its decompressor recovered them.

Counterpart in the training signal. Even though the training set has no tasks on PDE solvers, hardware emulators, or compressors, every task puts the model in the same position: a fairly precise and realistic specification, a hidden checker, and no answer key. Earning reward consistently requires generating one's own evidence that a result is correct, which is what these trajectories show.

7.3 A common thread

The four observed behaviors share a common structure. In each case, the base model solved the core problem but stopped short of implementing and establishing that the whole job was done: it declared completion without checking all requirements, tested what it had built rather than what was asked, changed what it should have preserved, or weakly verified its work. The trained model treats the specification as a contract to be implemented in full and its own work as a hypothesis to be tested against correct and valid assumptions. Together with the shorter trajectories of §6.3, this suggests a model that spends its steps differently, not one that just tries harder; for instance, the trained model can spend less time on a wrong interpretation, constructing fewer irrelevant tests.

8 Conclusion

RL on 1,700 expert-built coding tasks, using GSPO with LoRA, improved Kimi K2.7 on all five external benchmarks we evaluated, by 4.7 to 20.0 pp. Analyzing paired trajectories shows that the

base model's failures are often near-misses, while the trained model closes the last-mile gap and implements the same tasks end to end. We also identify core behavioral differences in the trained model's trajectories on the tasks the base model failed, and show how each of these corresponds to the training data and reward design.

References

Anthropic. Claude Code. https://github.com/anthropics/claude-code, 2025. Agentic coding tool; first released February 24, 2025. Documentation: https://code.claude.com/docs/en/overview. Accessed September 2026.

Artificial Analysis. Terminal-Bench v2.1 benchmark leaderboard. https://artificialanalysis.ai/evaluations/terminalbench-v2-1, 2026. Independent evaluation using the Terminus 2 harness; pass@1 averaged over 3 repeats per task. Accessed September 2026.

Shiyi Cao, Sumanth Hegde, Dacheng Li, Tyler Griggs, Shu Liu, Eric Tang, Jiayi Pan, Xingyao Wang, Akshay Malik, Graham Neubig, et al. SkyRL-v0: Train real-world long-horizon agents via reinforcement learning. https://novasky-ai.notion.site/skyrl-v0, 2025. NovaSky (UC Berkeley Sky Computing Lab) blog post, May 7, 2025. Code: https://github.com/NovaSky-AI/SkyRL. Accessed September 2026.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing SWE-bench Verified. https://openai.com/index/introducing-swe-bench-verified/, 2024. OpenAI blog post, August 13, 2024. Accessed September 2026.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V. Le, Sergey Levine, and Yi Ma. SFT memorizes, RL generalizes: A comparative study of foundation model post-training. *arXiv preprint arXiv:2501.17161*, 2025. URL https://arxiv.org/abs/2501.17161.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2205.14135.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025. URL https://arxiv.org/abs/2501.12948. Also published in Nature 645, 633–638 (2025).

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025. URL https://arxiv.org/abs/2509.16941. Public leaderboard: https://labs.scale.com/leaderboard/swe_bench_pro_public.

Rishi Desai, Jesse Hu, Joan Cabezas, Neel Harsola, Pratyush Shukla, Roey Ben Chaim, Adnan El Assadi, Omkaar Mukund Kamath, Fenil Faldu, Prannay Hebbar, et al. SWE-Marathon: Can agents autonomously complete ultra-long-horizon software work? *arXiv preprint arXiv:2606.07682*, 2026. URL https://arxiv.org/abs/2606.07682. Benchmark website: https://www.swe-marathon.org/.

Anisha Gunjal, Anthony Wang, Elaine Lau, Vaskar Nath, Yunzhong He, Bing Liu, and Sean Hendryx. Rubrics as rewards: Reinforcement learning beyond verifiable domains. *arXiv preprint arXiv:2507.17746*, 2025. URL https://arxiv.org/abs/2507.17746.

Harbor Framework Team. Terminus-2: Harbor's reference agent for terminal environments. https://www.harborframework.com/docs/agents/terminus-2, 2025. Accessed September 2026.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments. https://github.com/harbor-framework/harbor, 2026. Concept DOI 10.5281/zenodo.20953922. Documentation: https://www.harborframework.com. Accessed September 2026.

Yun He, Wenzhe Li, Hejia Zhang, Songlin Li, Karishma Mandyam, Sopan Khosla, Yuanhao Xiong, Nanshu Wang, Xiaoliang Peng, Beibin Li, et al. AdvancedIF: Rubric-based benchmarking and reinforcement learning for advancing LLM instruction following. *arXiv preprint arXiv:2511.10507*, 2025. URL https://arxiv.org/abs/2511.10507.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022. URL https://arxiv.org/abs/2106.09685.

Maggie Huan, Yuetai Li, Tuney Zheng, Xiaoyu Xu, Seungone Kim, Minxin Du, Radha Poovendran, Graham Neubig, and Xiang Yue. Does math reasoning improve general LLM capabilities? Understanding transferability of LLM reasoning. *arXiv preprint arXiv:2507.00432*, 2025. URL https://arxiv.org/abs/2507.00432.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. DeepSWE: Measuring frontier coding agents on original, long-horizon engineering tasks. https://deepswe.datacurve.ai/, 2026. Datacurve, May 26, 2026; v1.1 released June 14, 2026. Also arXiv:2607.07946. Code: https://github.com/datacurve-ai/deep-swe. Accessed September 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. *arXiv preprint arXiv:2504.07164*, 2025. URL https://arxiv.org/abs/2504.07164.

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

Kimi Team, Yifan Bai, Yiping Bao, Y. Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi K2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025. URL https://arxiv.org/abs/2507.20534.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, 2015. URL https://arxiv.org/abs/1412.6980.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tülu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024. URL https://arxiv.org/abs/2411.15124.

Kilian Lieret and Carlos E. Jimenez. mini-SWE-agent: The minimal AI software engineering agent. https://github.com/SWE-agent/mini-swe-agent, 2025. GitHub repository; first PyPI release July 2025. Accessed September 2026.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://arxiv.org/abs/1711.05101.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, et al. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling RL. https://www.together.ai/blog/deepswe, 2025. Agentica and Together AI blog post, July 2, 2025. Accessed September 2026.

Ryan Marten. Terminal-Bench 4.0. https://www.tbench.ai/news/terminal-bench-4-0, 2026. August 28, 2026. Removes 8 tasks from Terminal-Bench 3.0 and fixes 19. Accessed September 2026.

Ryan Marten, Alex Shaw, and Andy Konwinski. Terminal-Bench 3.0. https://www.tbench.ai/news/terminal-bench-3-0, 2026. July 30, 2026. Hosted by Harbor and the Laude Institute. Dataset: https://github.com/harbor-framework/terminal-bench (concept DOI 10.5281/zenodo.22105680, which resolves to the latest dataset version). Accessed September 2026.

Sushant Mehta, Liudas Panavas, Suhaas Garre, and Edwin Chen. ComplexConstraints and beyond: Expert rubrics for RLVR. *arXiv preprint arXiv:2606.09118*, 2026a. URL https://arxiv.org/abs/2606.09118. Accepted at the GEM Workshop at ACL 2026.

Sushant Mehta, Logan Ritchie, Suhaas Garre, Ian Niebres, Nick Heiner, and Edwin Chen. EnterpriseBench Corecraft: Training generalizable agents on high-fidelity RL environments. *arXiv preprint arXiv:2602.16179*, 2026b. URL https://arxiv.org/abs/2602.16179.

Sushant Mehta, Logan Ritchie, Liudas Panavas, and Edwin Chen. Cross-benchmark generalization in long-horizon agents. *arXiv preprint arXiv:2608.00181*, 2026c. URL https://arxiv.org/abs/2608.00181. Accepted at the COLM 2026 Workshop on Agent Behavior.

Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://arxiv.org/abs/2601.11868.

Miles Team. Miles: Enterprise-grade reinforcement learning for large-scale model post-training. https://github.com/radixark/miles, 2026. Forked from slime; announced November 19, 2025 (https://www.lmsys.org/blog/2025-11-19-miles/). Accessed September 2026.

NVIDIA. NeMo Megatron Bridge. https://github.com/NVIDIA-NeMo/Megatron-Bridge, 2025. GitHub repository; first PyPI release August 2025. Accessed September 2026.

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 (ICML)*, 2025. URL https://arxiv.org/abs/2412.21139.

Logan Ritchie, Sushant Mehta, Liudas Panavas, and Edwin Chen. Post-training on office work improves software engineering: A behavioral account of cross-domain transfer. *arXiv preprint arXiv:2608.01604*, 2026. URL https://arxiv.org/abs/2608.01604.

John Schulman and Thinking Machines Lab. LoRA without regret. https://thinkingmachines.ai/blog/lora/, 2025. Thinking Machines Lab: Connectionism, September 29, 2025. DOI 10.64434/tml.20250929. Accessed September 2026.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. URL https://arxiv.org/abs/1909.08053.

Ved Sirdeshmukh, Kaustubh Deshpande, Johannes Mols, Lifeng Jin, Ed-Yeremai Cardona, Dean Lee, Jeremy Kritz, Willow Primack, Summer Yue, and Chen Xing. MultiChallenge: A realistic multi-turn conversation evaluation benchmark challenging to frontier LLMs. *arXiv preprint arXiv:2501.17399*, 2025. URL https://arxiv.org/abs/2501.17399.

Joar Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward hacking. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://arxiv.org/abs/2209.13085.

Terminal-Bench Team. Terminal-Bench 2.1: A revision of Terminal-Bench 2.0 that fixes 28 tasks. https://www.tbench.ai/news/terminal-bench-2-1, 2026. May 6, 2026. Terminal-Bench 2.0 and Harbor were released November 7, 2025 (https://www.tbench.ai/news/announcement-2-0). Accessed September 2026.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://arxiv.org/abs/2407.16741.

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 (NeurIPS)*, 2025. URL https://arxiv.org/abs/2502.18449.

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. URL https://arxiv.org/abs/2405.15793.

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/2504.21798.

Feng Yao, Liyuan Liu, Dinghuai Zhang, Chengyu Dong, Jingbo Shang, and Jianfeng Gao. Your efficient RL framework secretly brings you off-policy RL training. https://fengyao.notion.site/off-policy-rl, 2025. Blog post, August 2025. Accessed September 2026.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. DAPO: An open-source LLM reinforcement learning system at scale. *arXiv preprint arXiv:2503.14476*, 2025. URL https://arxiv.org/abs/2503.14476.

Yang Yue, Zhiqi Chen, Rui Lu, Andrew Zhao, Zhaokai Wang, Yang Yue, Shiji Song, and Gao Huang. Does reinforcement learning really incentivize reasoning capacity in LLMs beyond the base model? In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2504.13837.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group Sequence Policy Optimization. *arXiv preprint arXiv:2507.18071*, 2025. URL https://arxiv.org/abs/2507.18071.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient execution of structured language model programs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2312.07104.

Zilin Zhu, Chengxing Xie, Xin Lv, and slime Contributors. slime: An LLM post-training framework for RL scaling. https://github.com/THUDM/slime, 2025. GitHub repository. Accessed September 2026.