Dockerless: Environment-Free Program Verifier for Coding Agents

Anonymous ACL submission

Abstract

Program verifiers play a central role in training coding agents, including selecting trajectories for supervised fine-tuning (SFT) and providing rewards for reinforcement learning (RL). Standard execution-based verification requires running unit tests inside per-repository environments such as Docker images, incurring substantial environment setup costs. We propose Dockerless, an environment-free agentic patch verifier that evaluates generated code patches without executing them. Rather than simply matching candidate patches to references, Dockerless judges patch correctness using evidence gathered through agentic repository exploration. On a verifier evaluation benchmark, Dockerless outperforms the strongest open-source verifier by 14.3 AUC points. Using Dockerless as both the SFT trajectory filter and the RL reward enables a fully environment-free post-training pipeline. The resulting model reaches 62.0%, 50.0%, and 35.2% resolve rate on SWE-bench Verified, Multilingual, and Pro, respectively. It surpasses the Qwen3.5-9B baseline by 2.4, 8.7, and 2.9 points, matching environment-based post-training.

1 Introduction

Program verifiers play a critical role in training automated coding agents. Whether curating high-quality trajectories for supervised fine-tuning (SFT) (Yuan et al., 2023; Pan et al., 2025; Jain et al., 2025) or providing rewards for reinforcement learning (RL) (Wei et al., 2026; Luo et al., 2025), verifiers determine whether the agent rollouts successfully resolve issues. Currently, the gold standard for this correctness feedback relies on executing test cases inside isolated, per-repository environments (Jimenez et al., 2024; Pan et al., 2025; Jain et al., 2025).

However, execution-based verification imposes substantial engineering overhead. Setting up these environments requires building custom Docker images, resolving per-repository dependencies, identifying relevant tests, and writing test-execution scripts and result parsers. Even advanced automated pipelines still succeed on only a limited share of candidate repositories (Jain et al., 2025; Badertdinov et al., 2026a; Li et al., 2026; Zhang et al., 2026b). More fundamentally, many real-world repositories, especially private, enterprise, or legacy codebases, lack reproducible environments or comprehensive test suites altogether (Pan et al., 2025; Zan et al., 2026), making execution-based verification unreliable or infeasible.

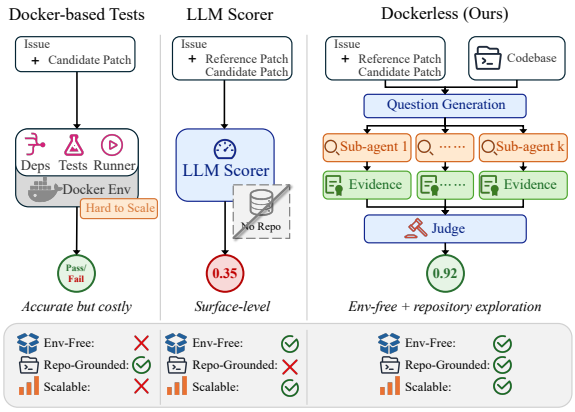

Figure 1: Comparison of verifiers for SWE agents. Docker-based tests are accurate but depend on costly per-repository environments. LLM scorers sidestep that cost but score patches based on surface-level information, without actively inspecting the repository. Dockerless instead deeply explores the codebase to judge the patch, requiring no per-repository environment while retaining repository grounding.

To reduce setup costs, recent work executes agent rollouts from a single shared base image rather than per-repository Docker containers (Sun et al., 2026; Ludwig et al., 2026; Xu et al., 2025). Yet, the verifier remains a critical bottleneck. Existing environment-free verifiers score patches using only surface-level information, without ever inspecting the repository (Shum et al., 2025; Wang

et al., 2026; Luo et al., 2025). Such shallow approaches are insufficient for complex SWE tasks, where determining functional equivalence requires deep repository context: for example, whether a modified function is actually called by the failing behavior, or whether an alternative implementation correctly integrates with surrounding modules.

To close this gap, we propose Dockerless, an environment-free agentic verifier that actively explores the repository to judge patch correctness. As shown in Fig. 1, rather than blindly matching textual diffs, Dockerless grounds its verification in the actual codebase. Given an issue description, a reference patch, and a candidate patch, Dockerless first derives several verification questions from the issue and the reference patch. It then dispatches dedicated sub-agents to gather repository evidence for each question. Finally, it aggregates the collected evidence into a correctness score indicating whether the candidate patch correctly resolves the issue. We train Dockerless via rejection sampling on 3.7K issues from SWE-Gym (Pan et al., 2025) and Multi-SWE-RL (Zan et al., 2026), retaining only question-answer-judge trajectories whose final verdict matches the ground-truth test outcome.

Ultimately, Dockerless unlocks a fully environment-free post-training pipeline: rollout collection, SFT data filtering, and RL reward computation can all run on a minimal base image with **zero per-repository setup**. As a standalone verifier, Dockerless outperforms the strongest open-source baseline by 14.3 AUC points on a verifier evaluation benchmark. For SFT, training on the top 25% of trajectories filtered by Dockerless (4K out of 16K) surpasses training on the full environment-free pool by 1.8, 6.4, and 3.4 points on SWE-bench Verified, Multilingual, and Pro, respectively. For RL, using Dockerless as an environment-free reward outperforms RL with the DeepSWE Verifier by 1.4, 2.7, and 1.1 points on the same three benchmarks. End-to-end, our fully environment-free post-training pipeline produces a model that reaches 62.0%, 50.0%, and 35.2% resolve rate on SWE-bench Verified, Multilingual, and Pro (Jimenez et al., 2024; OpenAI, 2024; Yang et al., 2026; Deng et al., 2025), improving over the Qwen3.5-9B baseline by 2.4, 8.7, and 2.9 points, respectively. By matching the performance of standard environment-based post-training, Dockerless establishes environment-free post-training as a scalable and viable path for the vast long tail of real-world repositories.

- We propose Dockerless, an environment-free agentic verifier that scores patches by actively exploring the repository with parallel sub-agents.
- By providing reliable correctness feedback, Dockerless enables a fully environment-free post-training pipeline for SFT trajectory filtering and RL rewards, scaling coding-agent post-training.
- Empirically, Dockerless outperforms the strongest open-source verifier by 14.3 AUC points, while the resulting fully environment-free post-training pipeline achieves performance comparable to standard environment-based post-training.

2 Methodology

2.1 Problem Setting

Given an issue x and a candidate patch y, a verifier assigns a correctness score $r(x, y) \in [0, 1]$ indicating whether y resolves x.

In standard SWE post-training, which we call the *environment-based* (*env-based*) setting, candidate patches are verified by executing held-out tests inside a repository-specific environment (E_x). E_x consists of a Docker image with pinned dependencies, a curated unit-test suite, and a working test runner. This produces a binary correctness signal:

$$r_{\text{env}}(x, y) = \mathbb{1}[\,\text{tests in } E_x \text{ pass under } y\,]. \quad (1)$$

However, building these environments is prohibitively expensive, and many real-world codebases lack reproducible environments or usable test suites.

To make post-training scalable, we consider the *environment-free* (*env-free*) setting in which agents run in a single minimal base image without repository-specific dependencies, test runners, or access to E_x. This setting is already practical on the agent side: frontier models under the OpenHands scaffold retain much of their performance after removing the per-repository environment, with resolve-rate drops of 3.0–13.9 points (Appendix A). Thus, env-free rollouts can be collected at scale; the remaining bottleneck is verification. Our goal is to train an environment-free verifier $r_\phi(x, y)$ that can replace r_{env} for both SFT trajectory filtering and RL reward computation.

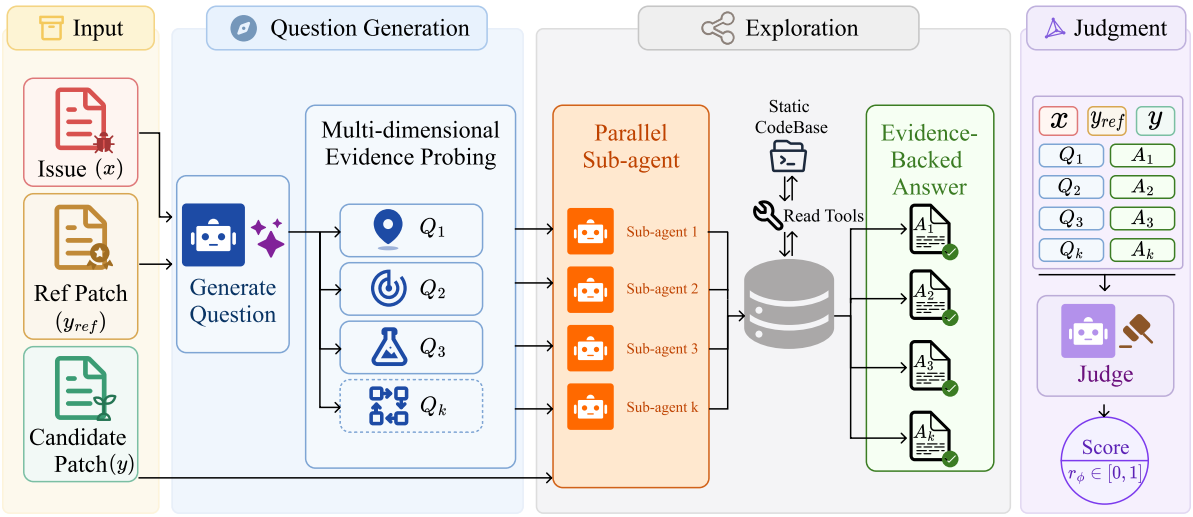

Figure 2: Architecture of Dockerless. The verifier takes the issue x, reference patch y_{ref}, and candidate patch y, and proceeds in two stages. (1) Question generation and exploration: the verifier first generates K verification questions and dispatches parallel sub-agents to collect evidence-backed answers from the codebase. (2) Judgment: the verifier conditions on the issue, the patches, and the collected (Q_k, A_k) pairs to produce a binary verdict token, whose logits define the continuous score $r_\phi(x, y)$.

2.2 Architecture of Dockerless

As illustrated in Fig. 2, the verifier operates in two stages. First, given an issue x and a reference patch y_{ref}, the model proposes a small set of verification questions $\{Q_1, \ldots, Q_K\}$. These questions ask, for example, *where in the repository the fix should take effect*, *what the patched code is supposed to do*, *what tests or assertions would confirm correctness*, and *whether other parts of the repository could break*. Answering these questions grounds the verifier's eventual judgment in repository exploration rather than in surface-level comparison between the candidate and the reference patch. For each question, a sub-agent then explores the repository through read-only shell tools (e.g., find, grep, rg) and returns a short evidence-backed answer A_k. The K sub-agents run in parallel for efficiency.

After all sub-agents return their answers, Dockerless aggregates the collected evidence to judge whether the candidate patch y resolves the issue x. Given $(x, y_{\text{ref}}, y, \{(Q_k, A_k)\}_{k=1}^{K})$, the verdict model outputs a binary token in $\{0, 1\}$, where 1 denotes a correct patch. At inference time, we convert the logits of the two verdict tokens into a continuous score:

$$r_\phi(x, y) = \frac{\exp(\ell_1)}{\exp(\ell_0) + \exp(\ell_1)},$$

where ℓ_0 and ℓ_1 are the logits for tokens 0 and 1. The full prompts used at both stages are listed in Appendix G.

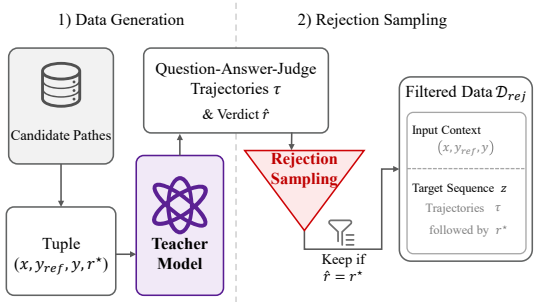

Figure 3: Training pipeline for Dockerless: teacher-generated question-answer-judge trajectories are rejection-sampled by matching the predicted verdict against the ground-truth, and used to fine-tune a base model.

2.3 Dockerless Training

We train the verifier r_ϕ via rejection sampling on execution-labeled candidate patches. Each example is a tuple $(x, y_{\text{ref}}, y, r^\star)$, where $r^\star \in \{0, 1\}$ is the ground-truth verdict obtained by running the held-out unit tests on the candidate patch y. Fig. 3 gives an overview of the training pipeline.

To construct training trajectories τ for the question-answer-judge, an agent powered with a teacher model explores the repository until making a judgment verdict $\hat{r} \in \{0, 1\}$ for each example. We then reject-sample these trajectories, keeping only those whose $\hat{r}$ matches the execution label $r^\star$; the retained examples form $\mathcal{D}_{\text{rej}}$. This keeps the training signal consistent end-to-

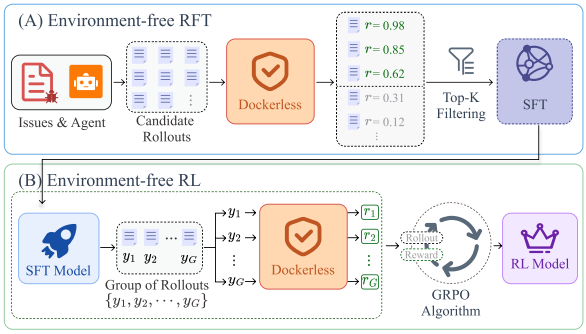

Figure 4: Env-free post-training pipeline for Dockerless. (A) Environment-free RFT: candidate rollouts are scored by Dockerless, and the top-K are kept to fine-tune the base model, yielding the SFT model. (B) Environment-free RL: starting from the SFT model, GRPO uses Dockerless as the per-rollout reward source, yielding the RL model.

end, and the verifier learns how to reason step-by-step and conclude the final verdict rather than from lucky matches. We additionally cap the negative-to-positive sample ratio at ρ to mitigate class imbalance, following the recipe of Shum et al. (2025).

The verifier is then trained with the standard next-token cross-entropy over the full output sequence.

$$\mathcal{L}_\phi = -\mathbb{E}_{\mathcal{D}_{\text{rej}}}\left[\sum_{t=1}^{T} \log p_\phi(z_t \mid x, y_{\text{ref}}, y, z_{<t})\right], \tag{2}$$

where $z = (z_1, \ldots, z_T)$ denotes the token sequence in the question-answer-judge trajectories τ. A single backbone is shared across question generation, sub-agent exploration, and the final judging stage, jointly trained under Eq. (2). Full training details are in Appendix D.1.

2.4 Environment-Free Post-training

With Dockerless trained, we now apply it in the environment-free post-training pipelines, illustrated in Fig. 4.

Environment-free RFT. Rejection-sampling fine-tuning (RFT) (Yuan et al., 2023) curates SFT data by keeping only the high-quality rollouts whose final patches pass per-repository unit tests (Pan et al., 2025; Jain et al., 2025). We instead start from an agent, collect a large pool of rollouts in a minimal Linux image without instantiating per-repository environments, and use Dockerless as the rejection signal. We score each rollout's final patch with Dockerless and form $\mathcal{D}_{\text{RFT}}$ by keeping the top-K rollouts globally ranked by r_ϕ. We then fine-tune the model on $\mathcal{D}_{\text{RFT}}$ with the standard SFT objective, yielding the SFT model.

Environment-free RL. We further use Dockerless as the reward model for RL on top of the SFT model. During RL, rollouts are collected in the same minimal Linux image used for env-free RFT, i.e., without a per-repository environment. For each rollout on issue x, let y_i denote its final patch. We score y_i with Dockerless and use $r_\phi(x, y_i)$ as the reward. We then optimize the model with GRPO (Shao et al., 2024). For each group of G rollouts on issue x, let $\{y_1, \ldots, y_G\}$ denote their final patches. We form group-normalized advantages

$$A_i = \frac{r_\phi(x, y_i) - \bar{r}}{\hat{\sigma}_r}, \qquad \bar{r} = \frac{1}{G}\sum_{j=1}^{G} r_\phi(x, y_j), \tag{3}$$

where $\hat{\sigma}_r$ is the standard deviation of the verifier rewards in the group. These advantages are then used in the standard GRPO objective. To improve reward stability, we compute each reward by averaging M independent Dockerless evaluations of the same final patch.

3 Experimental Settings

Benchmarks. For agent resolve rate, we evaluate on SWE-bench Verified (Jimenez et al., 2024; OpenAI, 2024), SWE-bench Multilingual (Yang et al., 2026), and SWE-bench Pro (Deng et al., 2025). For evaluating the verifier itself, we follow recent practice (Shum et al., 2025) and construct a balanced trajectory-level verifier evaluation benchmark of 776 samples (500 from SWE-bench Verified and 276 from Multi-SWE-bench Flash (Zan et al., 2026)); construction details are in Appendix C.3.

Evaluation protocol. We use OpenHands (Wang et al., 2025b) as the default agent scaffold with a maximum of 150 turns. For env-based evaluation, the agent runs inside the original per-repository Docker image with repository dependencies and test runners. For env-free evaluation, the agent runs in a minimal Ubuntu 22.04 LTS image (`ubuntu:jammy-20260109`) with only the repository checkout at the base commit. The main paper reports env-based evaluation, following the standard SWE evaluation protocol;

Model	Base	Training	Env-free	Verified	Multilingual	Pro
Open-source models						
SWE-Gym-7B	Qwen2.5-Coder-7B	SFT	No	11.4	3.0	3.3
SWE-Dev-7B	Qwen2.5-Coder-7B	SFT	No	10.6	4.2	4.0
SWE-Lego-8B	Qwen3-8B	SFT	No	41.2	19.0	16.1
Qwen3.5-9B	–	–	–	59.6	41.3	32.3
Ours: SFT on varying rollout sources						
Env-SFT-9B	Qwen3.5-9B	SFT	No	60.0	48.3	33.9
Dockerless-SFT-9B	Qwen3.5-9B	SFT	Yes	**60.6**	**47.7**	**35.3**
Ours: RL on varying rollout and reward sources						
+DeepSWE-Verifier RL	Dockerless-SFT-9B	RL	Yes	60.6	47.3	34.1
+Test-Execution RL	Dockerless-SFT-9B	RL	No	62.4	51.3	35.7
Dockerless-RL-9B	Dockerless-SFT-9B	RL	Yes	**62.0**	**50.0**	**35.2**

Table 1: Resolve rate (%) on SWE-bench Verified, Multilingual, and Pro under env-based evaluation. *Base* is the starting model of each training stage; *Training* marks SFT or RL; *Env-free* indicates whether the full training stage avoids per-repository Docker: "Yes" uses only a minimal base image, while "No" uses per-repository Docker; Bold rows are our headline models; gray rows are controlled comparisons that isolate the SFT rollout source or the RL rollout and reward source.

env-free numbers are deferred to Appendix B. We report resolve rate for issue resolution. For verifier evaluation, we follow (Shum et al., 2025) and report AUC, a discrimination metric aligned with SFT filtering and RL rewards.

Baselines. For agent performance, we compare our SFT and RL models against open-source SWE specialists at the same scale (under 10B parameters): SWE-Gym-7B (Pan et al., 2025), SWE-Dev-7B (Wang et al., 2025a), SWE-Lego-8B (Tao et al., 2026), and the base model Qwen3.5-9B (Qwen Team, 2026). For verifier evaluation, we compare Dockerless against four frontier LLMs used zero-shot as judges (DeepSeek-V3.2 (Liu et al., 2025), Kimi-K2.5 (Team et al., 2026), GLM-5 (Zeng et al., 2026), GPT-5.4 (OpenAI, 2026)) and four trained verifiers: SWE-Gym Verifier (Pan et al., 2025), R2E-Gym Verifier (Jain et al., 2025), OpenHands Critic (Wang et al., 2026), and DeepSWE Verifier (Luo et al., 2025).

Implementation details. We use Qwen3.5-9B (Qwen Team, 2026) as the backbone for both Dockerless and the downstream post-training. Dockerless is trained on rejection-sampled trajectories from 3.7K execution-labeled issues drawn from SWE-Gym (Pan et al., 2025) and Multi-SWE-RL (Zan et al., 2026), and uses $K=2-4$ verification questions. For downstream post-training, we use SWE-Rebench-v2 (Badertdinov et al., 2026b). We collect env-free trajectories for SFT and sample RL rollouts from the same task pool. Full training data construction and hyperparameters are deferred to Appendices C and D.

4 Results

4.1 Main results

Fully environment-free post-training reaches strongest open-source performance. Starting from Qwen3.5-9B, our fully environment-free post-training pipeline produces Dockerless-RL-9B, which reaches 62.0, 50.0, and 35.2 resolve rate on SWE-bench Verified, Multilingual, and Pro, respectively (Table 1). This improves over the base model by +2.4, +8.7, and +2.9 points and over the next-best open-source SWE specialist (SWE-Lego-8B) by +20.8, +31.0, and +19.1 points.

Env-free SFT matches env-based SFT. We next isolate the SFT stage by comparing two Qwen3.5-9B SFT models that differ only in the source of their training rollouts: Env-SFT-9B uses trajectories collected with a per-repository environment, while Dockerless-SFT-9B uses env-free trajectories filtered by Dockerless. Despite removing test execution from SFT data filtering, Dockerless-SFT-9B achieves comparable performance to the env-based baseline (60.6 vs. 60.0 on Verified, 47.7 vs. 48.3 on Multilingual, and 35.3 vs. 33.9 on Pro).

Model	Verified	Multi-SWE
DeepSeek-V3.2	69.4	58.5
Kimi-K2.5	70.7	63.9
GLM-5	73.2	62.5
GPT-5.4	75.9	59.5
SWE-Gym Verifier	61.0	53.7
R2E-Gym Verifier	64.3	55.1
OpenHands Critic	48.6	52.2
DeepSWE Verifier	66.7	62.9
Dockerless	**81.0**	**72.1**

Table 2: Verifier AUC on the trajectory-level verifier evaluation benchmark, with splits from SWE-bench Verified and Multi-SWE-bench Flash.

Env-free RL approaches env-based RL. We then isolate the RL stage on top of the same SFT initialization, Dockerless-SFT-9B. The three RL variants differ in their rollout environment and reward source: Dockerless-RL-9B uses env-free rollouts with Dockerless rewards, DeepSWE-Verifier RL uses DeepSWE Verifier rewards, and Test-Execution RL uses per-repository Docker with oracle test-execution rewards. Dockerless-RL-9B achieves performance close to Test-Execution RL (62.0 vs. 62.4 on Verified, 50.0 vs. 51.3 on Multilingual, and 35.2 vs. 35.7 on Pro), while outperforming DeepSWE-Verifier RL by $+1.4$, $+2.7$, and $+1.1$ points.

4.2 Verifier evaluation

We compare Dockerless against two baseline families on the balanced trajectory-level verifier evaluation benchmark (Table 2): four frontier LLMs (DeepSeek-V3.2, Kimi-K2.5, GLM-5, GPT-5.4) used zero-shot as judges (Appendix G.4), and four trained open-source verifiers (SWE-Gym Verifier, R2E-Gym Verifier, OpenHands Critic, DeepSWE Verifier).

With agentic repository exploration, Dockerless reaches 81.0 AUC on SWE-bench Verified and 72.1 AUC on Multi-SWE-bench Flash, outperforming every baseline in both splits. Compared with the strongest trained open-source verifier, Dockerless improves AUC by 14.3 points on Verified and 9.2 points on Multi-SWE-bench Flash; compared with the strongest frontier LLM judge, it improves by 5.1 and 8.2 points, respectively. These results show that the design of Dockerless, which combines agentic repository exploration with rejection-sampled trajectory training, yields a stronger patch verifier. This strong verifier performance is the key signal used in Section 4.1: Dockerless filters env-free SFT trajectories and supplies rewards for env-free RL, enabling post-training without per-repository test execution.

Training Data	Verified	Multilingual	Pro
None (base)	59.6	41.3	32.3
`All 16K`	58.8	41.3	31.9
`Random 4K`	58.2	44.3	32.0
`Env-based 4K`	60.0	**48.3**	33.9
`Dockerless 4K`	**60.6**	47.7	**35.3**

Table 3: Effect of the SFT data filter on downstream resolve rate (%) under env-based evaluation. All SFT rows use the same Qwen3.5-9B backbone and SFT recipe; only the selected training data differs. The base row reports Qwen3.5-9B.

4.3 Effect of the SFT data filter

We hold the SFT backbone and recipe fixed, and vary only the selected training data (Table 3). Starting from a pool of 16K env-free trajectories, `All 16K` trains on the full unfiltered pool, `Random 4K` samples 4K trajectories uniformly, and `Dockerless 4K` keeps the top-ranked 4K trajectories selected by Dockerless. As an env-based comparison, `Env-based 4K` uses 4K trajectories obtained with per-repository environments.

Dockerless achieves effective trajectory filtering. Training on all env-free trajectories does not improve over the base model: `All 16K` reaches 58.8, 41.3, and 31.9, below or equal to the base on all three benchmarks. This shows that raw env-free rollouts cannot be used directly for SFT; low-quality trajectories need to be filtered. `Dockerless 4K` substantially outperforms `Random 4K` on all three benchmarks (60.6 vs. 58.2 on Verified, 47.7 vs. 44.3 on Multilingual, and 35.3 vs. 32.0 on Pro), demonstrating that Dockerless provides a more effective selection signal than random sampling.

Env-free RFT matches env-based trajectory collection. More importantly, env-free trajectory collection combined with Dockerless filtering achieves performance comparable to SFT on env-based trajectories. `Dockerless 4K` matches `Env-based 4K` across the three benchmarks (60.6 vs. 60.0 on Verified, 47.7 vs. 48.3 on Multilingual, and 35.3 vs. 33.9 on Pro). This suggests a

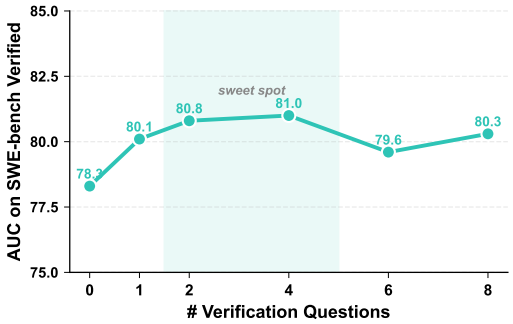

Figure 5: Verifier AUC vs. number of verification questions K on SWE-bench Verified verifier evaluation benchmark.

scalable path for RFT: collect rollouts without per-repository setup, then use a strong verifier to select the trajectories worth training on.

4.4 Effect of the number of verification questions

Fig. 5 studies how the number of verification questions affects Dockerless performance. We vary the number of verification questions $K \in \{0, 1, 2, 4, 6, 8\}$ on the SWE-bench Verified split of our verifier evaluation benchmark. For each setting, Dockerless first derives K verification questions from the issue and reference patch, dispatches one sub-agent per question to gather repository evidence, and then judges the candidate patch from the collected Q&A evidence. We report AUC against the execution-based ground truth.

Dockerless performance improves as K increases from 0 to 4, rising from 78.3 AUC with no verification question to 81.0 AUC at $K{=}4$. This shows that asking verification questions and gathering repository evidence helps Dockerless judge patch correctness. Beyond four questions, performance fluctuates rather than improving monotonically (79.6 at $K{=}6$, 80.3 at $K{=}8$), suggesting that additional questions often introduce redundant or noisy evidence. We therefore let Dockerless generate 2–4 verification questions at inference time, balancing verifier accuracy and per-call exploration cost.

4.5 Latency analysis

Dockerless performs a multi-step repository exploration before issuing a reward, so its reward computation is expected to take longer. We therefore analyze RL training latency under the three reward sources in Section 4.1: Dockerless, the DeepSWE Verifier, and `Test-Execution`, using 7680 rollouts.

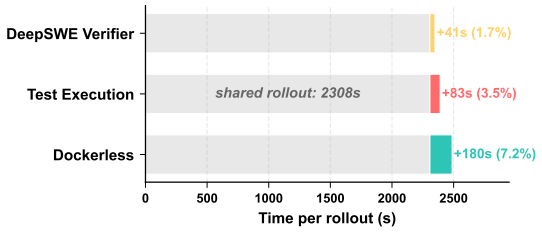

Figure 6: Per-rollout wall-clock breakdown during RL under three reward sources.

Fig. 6 decomposes each RL step into agent rollout time and reward-evaluation time. Agent rollouts dominate the wall-clock cost, taking 2308s on average, whereas reward evaluation adds only 41–180s. Although Dockerless requires more reward-evaluation time than the other verifier rewards, it still accounts for only 7.2% of the total per-rollout time. Thus, in the RL setting, the additional cost of agentic verification is small compared with the cost of generating rollouts. The end-to-end latency distribution shows the same pattern. As shown in Appendix F, total per-rollout times under the three reward sources almost overlap, because throughput is dominated by slow rollouts approaching the timeout rather than by reward computation.

4.6 Case study

Fig. 7 shows a representative case on a `matplotlib offsetText` color issue: the candidate patch passes execution ($r_{\text{env}}{=}1.0$) but rewrites the fix as an inline conditional rather than the helper-variable refactor used by the reference patch. Both baselines assign low scores: text similarity returns 0.468, and the DeepSWE Verifier assigns 0.035. Dockerless instead dispatches one sub-agent per verification question. The gathered evidence confirms that the fix is applied to both `XAxis` and `YAxis` initialization paths in `lib/matplotlib/axis.py`, and that the `inherit` vs. explicit `labelcolor` semantics are preserved. With this repository-grounded evidence, Dockerless scores the patch 0.996, in agreement with the execution result. The case illustrates how Q&A evidence can support a correct judgment even when the candidate patch differs substantially from the reference patch in surface form.

5 Related Work

5.1 Training methods for SWE agents

SWE agents are typically post-trained with a two-stage SFT-then-RL recipe on scaffolds such as SWE-agent (Yang et al., 2024) and Open-

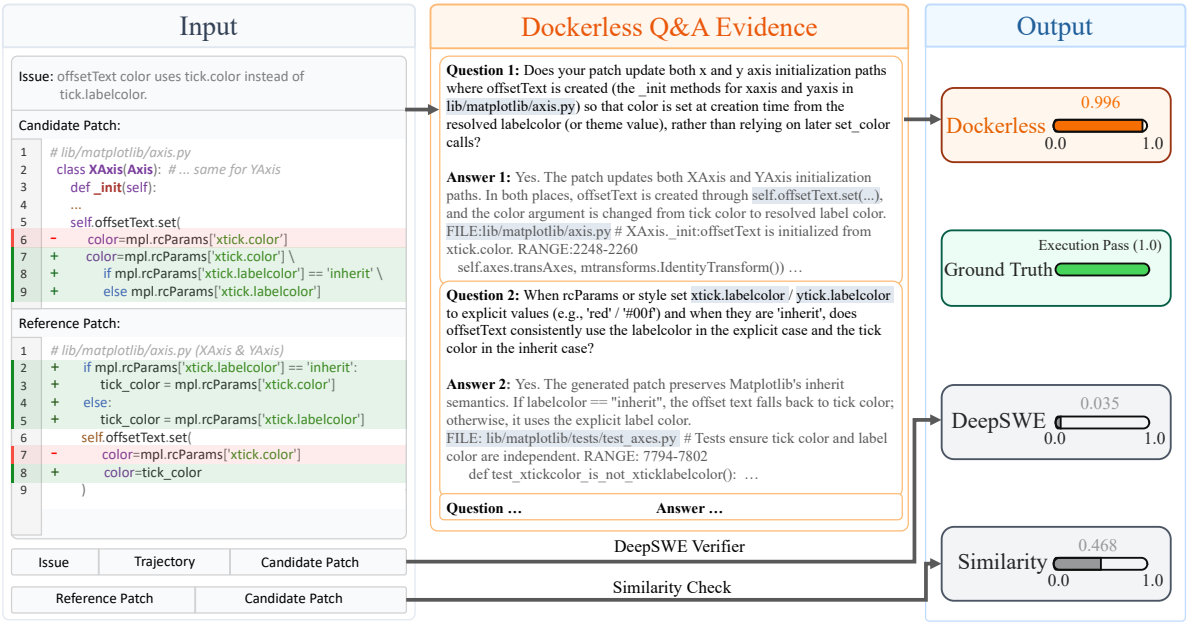

Figure 7: Representative case where the candidate patch resolves the issue but uses a different surface form from the reference patch.

Hands (Wang et al., 2025b), with SFT on curated or execution-filtered trajectories (Pan et al., 2025; Jain et al., 2025; Yang et al., 2026; Badertdinov et al., 2026a; Yang et al., 2025) and RL driven by test-execution rewards (Wei et al., 2026; Luo et al., 2025; Golubev et al., 2025; Shao et al., 2024; Yu et al., 2026). A complementary line builds env-free rollout pipelines that share a single base image across repositories (Sun et al., 2026; Xu et al., 2025; Ludwig et al., 2026), but they still constrain the agent during rollout, by exposing only a small set of static tools (Xu et al., 2025), by simulating tool returns with a learned transition model (Sun et al., 2026), or by prompt-level restrictions on what may be executed (Ludwig et al., 2026). Dockerless instead lets the agent issue any shell command in a minimal Linux image and receive real tool feedback, and replaces both the RFT-stage filter and the RL reward source with a single env-free agentic verifier.

5.2 Verifiers for SWE agents

A line of work trains LM verifiers that score a patch from a fixed prompt, ranging from execution-trained classifiers (Pan et al., 2025; Jain et al., 2025) and a scaled 30B mixture-of-experts critic (Shum et al., 2025) to group-wise textual reasoning over candidates (Xu et al., 2025) and rubric-supervised or RL-distilled variants (Wang et al., 2026; Luo et al., 2025). None of these call tools or inspect the repository at scoring time.

A more recent line frames the verifier itself as an agent, but places that agent outside the SWE patch outcome setting. Some target domains far from SWE patches, namely mathematical reasoning (Zhang et al., 2026a) and competitive programming (Ma et al., 2026). Others place the agent at rubric authoring rather than at scoring (Raghavendra et al., 2026), or score intermediate trajectory steps under a fixed rubric rather than the final patch (Han et al., 2026). Dockerless instead places the agent at SWE patch outcome scoring itself, actively exploring the repository through real tool calls before issuing a verdict.

6 Conclusion

In this work, we propose Dockerless, an agentic verifier that scores patches by actively exploring the repository, requiring no per-repository environment. We show that Dockerless can serve as both the trajectory filter for SFT and the reward signal for RL, yielding a fully environment-free post-training pipeline for coding agents. Dockerless outperforms prior open-source verifiers, and the resulting model matches the performance of its environment-based counterpart while requiring zero per-repository setup. We believe that agentic, evidence-grounded verification provides a new perspective on reward modeling for code, and opens a scalable path toward post-training on the long tail of real-world repositories without reproducible execution environments.

Limitations

The main limitation of Dockerless is latency. Because each scoring call performs multi-step repository exploration before issuing a verdict, it is inherently slower than verifiers that score from a single forward pass, averaging 180 seconds per call in our implementation. In our env-free RL setting this overhead is largely absorbed by the much longer rollout phase (Section 4.5), so it does not slow down end-to-end training. For use cases where the verifier is called outside this context, such as synchronous test-time critique or interactive IDE feedback, the latency may become a bottleneck. Mitigating this through caching, batching, or distillation into a faster scorer is straightforward, but we leave a systematic study to future work.

References

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2026a. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *Advances in Neural Information Processing Systems*, 38.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. 2026b. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, and 1 others. 2025. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, and 1 others. 2025. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*.

Hao Han, Jin Xie, Xuehao Ma, Weiquan Zhu, Ziyao Zhang, ZhiLiang Long, Hongkai Chen, and Qingwen Ye. 2026. Swe-trace: Optimizing long-horizon swe agents through rubric process reward models and heuristic test-time scaling. *arXiv preprint arXiv:2604.14820*.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. 2025. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights swe agents. *arXiv preprint arXiv:2504.07164*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157.

Kenan Li, Rongzhi Li, Linghao Zhang, Qirui Jin, Liao Zhu, Xiaosong Huang, Geng Zhang, Yikai Zhang, Shilin He, Chengxing Xie, and 1 others. 2026. Repolaunch: Automating build&test pipeline of code repositories on any language and any platform. *arXiv preprint arXiv:2603.05026*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Nikolai Ludwig, Wasi Uddin Ahmad, Somshubra Majumdar, and Boris Ginsburg. 2026. From swe-zero to swe-hero: Execution-free to execution-based fine-tuning for software engineering agents. *arXiv preprint arXiv:2604.01496*.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. 2025. DeepSWE: Training a fully open-sourced, state-of-the-art coding agent by scaling rl. https://www.together.ai/blog/deepswe.

Zeyao Ma, Jing Zhang, Xiaokang Zhang, Jiaxi Yang, Zongmeng Zhang, Jiajun Zhang, Yuheng Jing, Lei Zhang, Hao Zheng, Wenting Zhao, and 1 others. 2026. Scaling agentic verifier for competitive coding. *arXiv preprint arXiv:2602.04254*.

OpenAI. 2024. Introducing swe-bench verified.

OpenAI. 2026. Introducing GPT-5.4.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. 2025. Training software engineering agents and verifiers with swe-gym. In *International Conference on Machine Learning*, pages 47717–47737. PMLR.

Qwen Team. 2026. Qwen3.5: Towards native multimodal agents.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. 2026. Agentic rubrics as contextual verifiers for swe agents. *arXiv preprint arXiv:2601.04171*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical

reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

KaShun Shum, Binyuan Hui, Jiawei Chen, Lei Zhang, Jiaxi Yang, Yuzhen Huang, Junyang Lin, Junxian He, and 1 others. 2025. Swe-rm: Execution-free feedback for software engineering agents. *arXiv preprint arXiv:2512.21919*.

Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, and 1 others. 2026. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, and 1 others. 2026. Swe-lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*.

Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, and 1 others. 2026. Kimi k2. 5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*.

Haoran Wang, Zhenyu Hou, Yao Wei, Jie Tang, and Yuxiao Dong. 2025a. Swe-dev: Building software engineering agents with training and inference scaling. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 3742–3761.

Xingyao Wang, Valerie Chen, Heng Ji, and Graham Neubig. 2026. A rubric-supervised critic from sparse real-world outcomes. *arXiv preprint arXiv:2603.03800*.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, and 1 others. 2025b. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pages 65882–65919.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. 2026. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *Advances in Neural Information Processing Systems*, 38:78500–78525.

Junjielong Xu, Boyin Tan, Xiaoyuan Liu, Chao Peng, Pengfei Gao, and Pinjia He. 2025. Scalable supervising software agents with patch reasoner. *arXiv preprint arXiv:2510.22775*.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652.

John Yang, Kilian Lieret, Carlos Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. 2026. Swe-smith: Scaling data for software engineering agents. *Advances in Neural Information Processing Systems*, 38.

Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, and 1 others. 2025. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2026. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244.

Zheng Yuan, Hongyi Yuan, Chengpeng Li, Guanting Dong, Keming Lu, Chuanqi Tan, Chang Zhou, and Jingren Zhou. 2023. Scaling relationship on learning mathematical reasoning with large language models. *arXiv preprint arXiv:2308.01825*.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, and 1 others. 2026. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38.

Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, and 1 others. 2026. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*.

Jiazheng Zhang, Ziche Fu, Zhiheng Xi, Wenqing Jing, Mingxu Chai, Wei He, Guoqiang Zhang, Chenghao Fan, Chenxin An, Wenxiang Chen, and 1 others. 2026a. Agentv-rl: Scaling reward modeling with agentic verifier. *arXiv preprint arXiv:2604.16004*.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, and 1 others. 2026b. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38.

A Frontier-model env-base vs. env-free

Setting. We motivate Dockerless by first checking whether env-free agent rollouts are useful at all on SWE-bench. Four frontier models (DeepSeek-V3.2, Kimi-K2.5, GLM-5, GPT-5.4) are run under the OpenHands scaffold on SWE-bench Verified, Multilingual, and Pro. The env-based setting uses the per-repository Docker image with the held-out test suite, exactly as in Table 1; the env-free setting replaces it with the minimal Ubuntu image from Section 3, with no per-repository dependencies and test runner. We compare resolve rate (Pass@1) under the two settings.

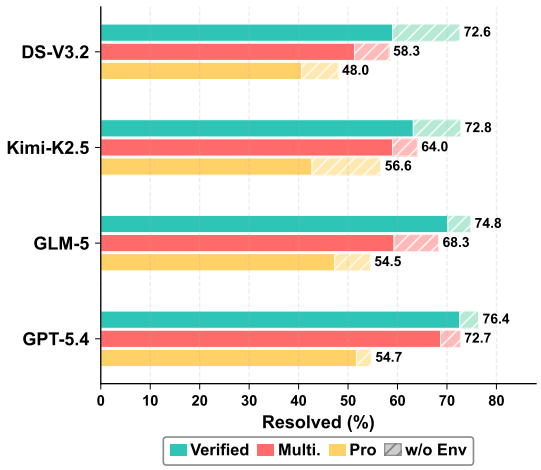

Figure 8: Frontier-model resolve rate (%) on SWE-bench Verified, Multilingual, and Pro under env-based and env-free settings. Solid bars are env-free; hatched extensions show the additional gain from per-repository environments, so the full bar height equals the env-based score.

Removing the environment costs only a few points. The hatched portion of every bar in Fig. 8 is small: across the four models and three benchmarks, env-free evaluation costs at most 13.9 points and on average 7.1 points of resolve rate, with the strongest model (GPT-5.4) staying within 3.0–4.0 points across all three benchmarks. In the env-free setting, the model already solves a large fraction of issues that its env-based counterpart can solve on the same benchmarks.

Implications for the verifier. The agent side of the env-free pipeline is therefore already largely feasible: rollouts can be collected at scale on benchmarks where no per-repository Docker is available, with only a moderate quality hit. The blocker is the verifier side, since without test execution there is no built-in correctness signal to filter rollouts or reward RL. This is the gap Dockerless closes; the corresponding env-free results on our own models are reported in Appendix B.

Model	Verified	Multilingual	Pro
SWE-Gym-7B	9.0	5.7	2.2
SWE-Dev-7B	8.4	7.0	2.2
SWE-Lego-8B	32.0	17.7	12.7
Qwen3.5-9B	50.2	38.3	26.0
Env-SFT-9B	50.0	36.7	27.2
Dockerless-SFT-9B	**52.0**	**41.0**	**29.4**
Dockerless-RL-9B	**53.8**	**42.3**	**30.6**

Table 4: Resolve rate (%) on SWE-bench Verified, Multilingual, and Pro under env-free evaluation: the agent runs in a minimal Ubuntu image with only the repository checkout, with no per-repository Docker image and no pre-installed dependencies. These are the same models as Table 1, evaluated under the stricter env-free setting.

B Env-free evaluation results

Setting. Table 1 reports env-based evaluation only. For completeness, we re-evaluate the same models from Table 1 under env-free evaluation, where the agent runs in a minimal Ubuntu 22.04 LTS image with only the repository checkout at the base commit, with no per-repository Docker, no test runner, and no pre-installed dependencies (definition in Section 3). Numbers are reported in Table 4.

Ranking is preserved across environments. Dockerless-RL-9B remains the strongest sub-10B model on every benchmark under env-free evaluation (53.8, 42.3, 30.6), ahead of Dockerless-SFT-9B by 1.2–1.8 points and of the env-based-SFT baseline Env-SFT-9B by 3.8, 5.6, and 3.4 points on Verified, Multilingual, and Pro respectively. The aggregate story from Table 1 therefore carries over to env-free deployment: a fully env-free pipeline (Dockerless-SFT-9B and Dockerless-RL-9B trained without test execution) still produces the strongest model when the deployment setting also forbids test execution.

Dockerless-trained models are more robust to env-free deployment. Comparing each model's env-based score in Table 1 to its env-free score in Table 4, the average drop is 9.4 points for Env-SFT-9B, 7.1 points for Dockerless-SFT-9B, and 6.8 points for Dockerless-RL-9B. The env-

free-trained models thus suffer a smaller env-base-to-env-free gap than the env-based-trained baseline, which is the expected direction: a model that has been trained on env-free rollouts has seen the same distribution it is evaluated on at deployment, while the env-based-trained baseline is exposed to a distribution shift. The same trend holds for the open-source SFT specialists, whose absolute scores are too low to draw strong conclusions but whose gaps lie in the same range.

C Dataset Construction

C.1 Agentic verifier training data

We construct a training corpus from execution-labeled patches in SWE-Gym (Pan et al., 2025) and Multi-SWE-RL (Zan et al., 2026), with $r^\star \in \{0, 1\}$ being the verdict obtained from running the held-out unit tests on the candidate patch. These datasets are disjoint from our verifier evaluation benchmark built from SWE-bench Verified and Multi-SWE-bench Flash. For each source example, a strong frontier teacher model (GLM-5) proposes one or more candidate (Q+A+Judge trajectory, $\hat{r}$) tuples via the same workflow used at inference (Section 2.2). We then keep only tuples whose predicted verdict $\hat{r}$ matches $r^\star$, and apply two additional cleaning passes: we discard answer trajectories with fewer than 4 or more than 30 turns, and we remove malformed or interrupted exchanges. Finally, we cap the negative-to-positive sample ratio at 4:1 to mitigate class imbalance, following the recipe of Shum et al. (2025). The resulting corpus covers 3.7K unique issues. Each training example bundles one question-generation trajectory, K sub-agent answer trajectories, and one final-judgment trajectory, all generated by the same teacher. We do not enforce a target ratio across these three sub-tasks; they are jointly trained on the same backbone under Eq. (2). Both inputs (the candidate patch text and the rendered Q+A context) are truncated to 10,000 characters before being fed to the model.

C.2 Env-free rollout data

The pool of env-free rollouts used to construct $\mathcal{D}_{\text{RFT}}$ is collected on SWE-Rebench-v2 (Badertdinov et al., 2026b). Starting from the OpenHands agent and a minimal Linux image, the agent receives only the issue x and the repository at the base commit, with no per-repository Docker image; OpenHands tools remain available and may include execution feedback from running standard developer utilities. We collect a pool of 16K rollouts at sampling temperature 1.0, from which the downstream filter selects 4K trajectories globally ranked by Dockerless.

C.3 Verifier evaluation benchmark

We construct a balanced trajectory-level verifier benchmark to evaluate Dockerless against prior verifiers (Table 2). The benchmark contains 500 samples drawn from SWE-bench Verified and 276 samples drawn from Multi-SWE-bench Flash, with positive and negative labels balanced within each split. Trajectories are collected from several models running under the SWE-agent and OpenHands scaffolds in a 1:1 split, and each (issue, candidate patch) pair is labeled positive or negative via standard evaluation inside the per-repository Docker environment with held-out tests; positive and negative labels are balanced 1:1 within each split.

D Training Details

D.1 Agentic verifier

The agentic verifier is initialized from Qwen3.5-9B (Qwen Team, 2026) and fine-tuned with standard next-token cross-entropy on the filtered trajectories described in Appendix C.1 (Eq. (2)). We report results from the best checkpoint, reached at 150 optimizer steps on a held-out validation split. We use AdamW with learning rate $1.0\text{e}{-5}$ (cosine decay to $1.0\text{e}{-6}$, warmup ratio 0.05), weight decay 0.01, batch size 256, and maximum sequence length 32,768.

At inference, we serve the trained model via vLLM with the OpenAI-compatible API. The verifier generates 2–4 verification questions per scoring call, with one sub-agent dispatched per question to explore the repository in parallel. At the answer position, we read the logits of the "0" and "1" verdict tokens and convert them into the dense score via softmax (Section 2.2).

D.2 Env-free SFT

Each candidate rollout's final patch is scored by Dockerless with $M{=}2$ independent agentic passes; we report the mean dense score and discard any pass that fails (e.g., due to inference-time errors or timeouts). We then build $\mathcal{D}_{\text{RFT}}$ by selecting the top-ranked 4K rollouts globally from

the 16K pool. The SFT model is initialized from Qwen3.5-9B (Qwen Team, 2026) and trained with standard maximum-likelihood on $\mathcal{D}_{\text{RFT}}$. We use the same AdamW configuration as the verifier (Appendix D.1), trained for 3 epochs.

D.3 Env-free RL

We initialize the RL policy from the SFT model produced in Appendix D.2 and run GRPO (Shao et al., 2024) with Dockerless as the per-rollout reward source. For each issue x, we sample a group of $G{=}8$ rollouts and score every rollout with $M{=}2$ independent agentic passes through Dockerless; failed passes are dropped and the remaining dense scores are averaged to form $r_\phi(x, y_i)$. The group-normalized advantages from Eq. (3) drive the policy update following the standard GRPO objective; no test execution is performed at any step. We use actor learning rate $2.0\text{e}{-6}$, training batch size 64, PPO mini-batch size 64, 8 responses per prompt, clipping range $[0.2, 0.27]$, entropy coefficient 0, KL coefficient 0 (no KL loss), maximum 150 turns per rollout, and sampling temperature 1.0. We train for 50 RL steps in total.

E Per-Language Analysis

While the aggregate SFT (w/o env) vs. SFT (w/ env) gap in Table 1 is small, the per-language breakdown (Fig. 9) is uneven, and the unevenness has a consistent pattern.

The two languages where SFT (w/ env) clearly wins are also the two compilation-heavy ones in the benchmark: Rust (+7.0) and C (+13.3). On the remaining high-volume languages (Python, Go, JavaScript, Java, PHP), the two settings stay within ± 2.5 points of each other. The two large gaps below the diagonal (TypeScript -13.3, C++ -8.3) come from splits with only 30 and 12 instances, so we do not read them as evidence either way.

We attribute the Rust/C gap to compiler diagnostics being available only inside the per-repository environment: env-base trajectories can observe type errors and link failures at intermediate steps, while env-free trajectories must infer the same information from the source alone. This is consistent with the broader claim that the residual value of env-base supervision is concentrated in compiler signal, not in test execution per se, although Rust and C are only two languages and we do not treat this as proof.

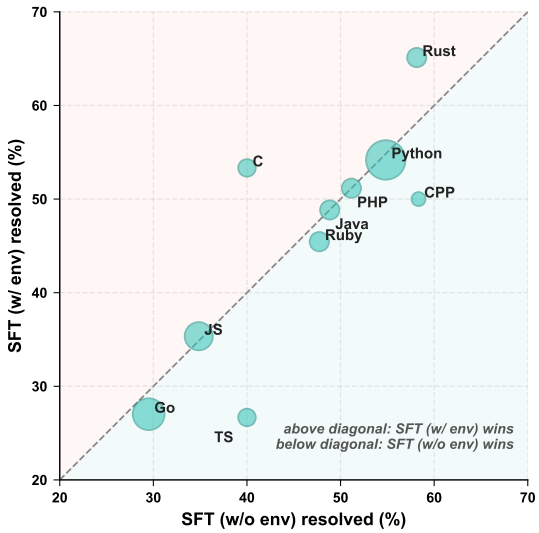

Figure 9: Per-language comparison of SFT (w/o env) and SFT (w/ env) resolve rate, aggregated across SWE-bench Verified, Multilingual, and Pro. Bubble size encodes the number of test instances per language. Points above the diagonal mark languages where SFT (w/ env) wins, below the diagonal where SFT (w/o env) wins.

The takeaway for Table 1 is that the headline "env-free matches env-base" holds across the high-volume languages that dominate the aggregate, but understates a real 7–13 point gap on compilation-heavy languages. Closing that gap likely requires surfacing compiler-style feedback inside the env-free pipeline rather than scaling env-base data further, which we leave to future work.

F Latency distribution

Figure 10 complements the mean numbers in Section 4.5 by showing the full distribution of per-rollout wall-clock time (rollout + reward) for the three reward sources. The three distributions overlap almost completely: a single mode around 2400–3000s and a heavy tail extending to the hard timeout. The choice of reward source shifts the mean by less than 150s, well inside the spread of the rollout distribution itself, so the end-to-end RL step is bottlenecked by the slowest rollouts in each group rather than by reward latency.

G Prompt Templates

We list below the full prompts used by Dockerless.

G.1 Question generation prompt

The question generator takes the issue description and the reference patch and emits 2–4 diagnostic questions, each tagged with one of four categories

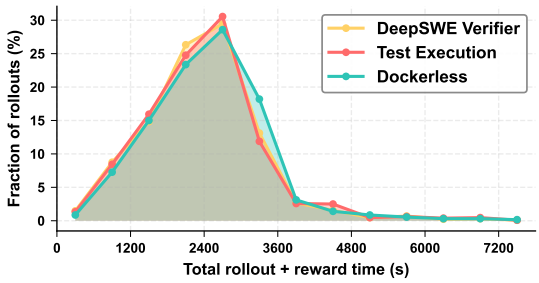

Figure 10: Distribution of total per-rollout wall-clock time (rollout + reward) under three reward sources, on 7680 rollouts collected during RL training. All three sources produce near-identical distributions; the long right tail is set by slow rollouts, not by the verifier.

(location, behavior, test evidence, edge case) and accompanied by a short rationale.

Stage 1: Question Generation Prompt

```
## Role
You are an expert code reviewer specialized in analyzing patches across different programming languages.
## Task
Generate 2-4 focused questions that would provide strong evidence for judging whether a candidate's patch correctly fixes the issue. Your questions should help distinguish a correct fix from an incorrect or incomplete one.
## Input
### Issue Description
{issue_description}
### Golden Patch (Correct Solution)
{golden_patch}
## Analysis Framework
Before generating questions, briefly analyze: (1) Bug Type, (2) Failure Scenario, (3) Success Criteria.
## Question Categories
Your questions MUST fall into one of these categories:
  • Location: identify WHERE the fix should be applied.
  • Behavior: understand WHAT the code should do.
  • Test Evidence: find HOW to verify correctness.
  • Edge Cases: discover WHAT ELSE might break.
## Output Format
Output a single JSON object with an analysis field (bug_type, failure_scenario, success_criteria) and a questions list, each item carrying id, category, question, and rationale.
## Key Principle
Focus on evidence that can distinguish between a complete fix, a partial fix, and an incorrect fix.
```

G.2 Sub-agent exploration prompt

The sub-agent runs a ReAct-style read-only shell loop. It is configured with a fixed system message and a per-question instance template, and emits a final answer through a SUBMIT_ANSWER heredoc.

Stage 1: Sub-agent System Message

```
You are a code exploration assistant that answers questions by running shell commands. You analyze codebases in various programming languages (Python, Java, JavaScript, Go, C++, etc.).
## Response Format
Every response MUST contain: (1) a THOUGHT section explaining your reasoning, and (2) exactly ONE bash command in a code block.
## Rules
This is a READ-ONLY task; do NOT modify any files. Execute ONE command per response. Use non-interactive commands only. Chain commands with && or || when needed in a single block.
```

Stage 1: Sub-agent Per-Question Instance Template

```
<question>
Answer the following question by exploring the codebase.
### Issue Description
{issue_description}
### Golden Patch
{golden_patch}
### Question
{question}
</question>
<instructions>
## Goal
Find concrete evidence that helps verify if a patch correctly fixes the issue. This is a READ-ONLY task.
## Search Strategy (priority order)
(1) Test Evidence: find tests that validate the correct behavior. (2) Error Location: find where the error or failure occurs. (3) Code Context: understand the code being patched, including definitions and call sites. (4) Documentation & Configuration: find expected-behavior documentation.
## Final Answer Format
When evidence is sufficient, submit the answer via a cat «'ANSWER_EOF' block containing: a SUBMIT_ANSWER marker, a Direct Answer, Key Findings, and one or more FILE / RANGE code-evidence blocks.
## Important Notes
Focus on concrete evidence (actual code, tests, assertions). Prioritize test cases. Include file paths and line numbers for all evidence. Be language-aware but use language-agnostic search patterns.
</instructions>
```

G.3 Final scoring prompt

The judge model conditions on the issue, the reference patch, the candidate patch, and the Q&A context collected by the sub-agents, and produces a single binary verdict token whose logits define the continuous score $r_\phi(x, y)$.

Stage 2: Final Scoring Prompt

```
## Role:
You are an expert code reviewer.
## Task:
Evaluate whether the "Generated Patch"
solves the issue described in the "Issue
Description" correctly, using the Q&A context
gathered in Stage 1 and the golden patch (a
correct solution) as a reference.
## Issue Description:
{issue_description}
{context}
## Golden Patch:
{golden_patch}
## Generated Patch:
{generated_patch}
## Response Format:
You must follow the response format strictly:
<answer>[0 for not solving the issue, 1 for
solving the issue]</answer>
```

G.4 Zero-shot LLM-as-judge prompt

For the frontier LLM-as-judge baselines in Table 2 (DeepSeek-V3.2, Kimi-K2.5, GLM-5, GPT-5.4), we query each model zero-shot with the issue description, the reference (golden) patch, and the candidate patch, and parse the binary verdict from the `<answer>` tag.

Zero-shot LLM-as-Judge Prompt

```
## Role:
You are an expert code reviewer.
## Task:
Your task is to evaluate whether the
"Generated Patch" solves the issue described
in the "Issue Description" correctly based on
the issue description and a golden patch (a
reference correct solution).
## Issue Description:
{issue_description}
## Golden Patch:
{golden_patch}
## Generated Patch:
{generated_patch}
## Response Format:
You must follow the response format strictly:
<answer>[0 for not solving the issue, 1 for
solving the issue]</answer>
```