Advancing Environment Setup LLMs through Online Reinforcement Learning

Alexander Kovrigin[1,2], **Aleksandra Eliseeva**[1], **Konstantin Grotov**[1*], **Egor Bogomolov**[1,3], **Yaroslav Zharov**[1]
[1]JetBrains Research, [2]Constructor University, [3]Delft University of Technology

Abstract

Environment setup—the process of configuring the system to work with a specific software project—represents a persistent challenge in Software Engineering (SE). Automated environment setup methods could assist developers by providing fully configured environments for arbitrary repositories without manual effort. This also helps SE researchers to scale execution-based benchmarks. However, recent studies reveal that even state-of-the-art Large Language Models (LLMs) achieve limited success on automating this task. To address this limitation, we employ an online Reinforcement Learning with Verifiable Rewards approach to improve the environment setup capabilities of LLMs. As outcome-based rewards for environment setup require containerisation of each sample and are computationally expensive, we leverage lightweight proxy rewards. On EnvBench-Python, our method enables Qwen3-8B (a model runnable on consumer hardware) to set up 15.8 out of 329 repositories on average over five runs. This is a +690% gain over the base model and +58% over GPT-4o-mini at comparable cost. Our replication package with training code and trained model checkpoints is available online: `https://github.com/envsetup-rl-dl4c/envsetup-rl`.

1 Introduction

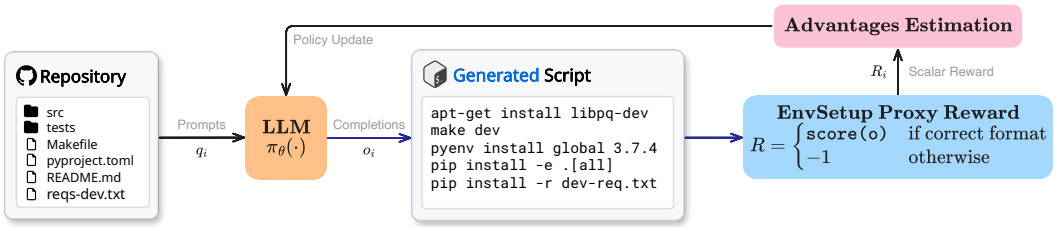

Figure 1: Overview of the proposed training pipeline. For i-th training sample, which is an individual repository, LLM π_θ is provided with the prompt q_i that contains the task description and the repository context, and it generates a completion o_i that is expected to contain a shell script. LLM response o_i is passed to a rule-based reward function R that outputs a float score R_i. REINFORCE++ algorithm is used to update the LLM weights given the reward scores R_i and the responses o_i.

Large Language Models (LLMs) show great promise for Software Engineering (SE) tasks [Liu et al., 2024]. While closed-source general-purpose models largely dominate benchmarks [Jain et al., Jimenez et al., 2024], open-source models remain strong competitors [DeepSeek-AI, 2025, Qwen Team, 2025, Kimi Team et al., 2025]. Recent studies demonstrate that task-specific autonomous agents

*Corresponding author to `konstantin.grotov@jetbrains.com`.

Deep Learning For Code in the Agentic Era, NeurIPS 2025.

powered by open-source models can solve various SE problems, including code generation [Hasan et al., 2025], bug localization [Ma et al., 2025, Chang et al., 2025, Reddy et al., 2025, Chen et al., 2025], and issue resolution [Luo et al., 2025, Wang, 2025, Pan et al., 2025, Zeng et al., 2025, Ma et al., 2025, Chang et al., 2025].

A common strategy for developing capable task-specific agents is to train them on carefully curated data [Pan et al., 2025, Zeng et al., 2025]. However, in the SE domain, the bottleneck has shifted from sophisticated data filtering strategies to acquiring sufficient data in the first place. Since agents operate in an interactive manner, this requires scaling the construction of interactive environments. This, in turn, often requires appropriately configuring the system to be able to execute the sample code. In this paper, we will call this configuration process an environment setup.

This limitation has far-reaching implications for SE benchmarks. For instance, SWE-Bench [Jimenez et al., 2024], one of the leading benchmarks for SE agents, includes only 12 Python repositories, and collecting and maintaining it required substantial manual effort. Scaling such datasets typically relies on manual setup [Pan et al., 2025] or on synthetic augmentation [Pham et al., 2025], trading realism for scale. Automated environment setup methods [Guo et al., 2025, Badertdinov et al., 2025, Zhang et al., 2025, Vergopoulos et al.] promise scalability with real data but remain limited—for instance, SWE-Rebench [Badertdinov et al., 2025] reports a 31% success rate on Python repositories overall, while on EnvBench [Eliseeva et al., 2025], a recently introduced benchmark for environment setup specializing on hard repositories, the best result is 6.69% of 329, achieved by GPT-4o in an agentic workflow.

Hence, our work focuses on advancing the environment setup capabilities of current LLMs. We analyze the environment setup scripts produced by strong LLMs on EnvBench and catalogue locally verifiable failure patterns. In the absence of a robust teacher for supervised learning, we adopt Reinforcement Learning with Verifiable Rewards (RLVR) [Lambert et al., 2024]. The proposed framework, presented in Figure 1, relies on rule-based rewards to improve the models' capabilities. To avoid slow and costly containerised execution of each sample, we introduce three lightweight proxy rewards. (i) *LLM-as-a-Judge* is an LLM-based feedback on environment setup script correctness; (ii) *Heuristics* is a set of deterministic rules that detect common environment setup fault patterns via static analysis; and (iii) *ShellCheck* is a general-purpose shell script quality check based on static analysis. Using these lightweight rewards enables online RL over hundreds of real repositories.

The Qwen3-8B model trained with the proposed approach consistently achieves better results on EnvBench-Python compared to the base model, showing the effectiveness of RLVR with designed proxy rewards for the environment setup task. Our best checkpoint trained with LLM-as-a-Judge reward successfully sets up 15.8/329 repositories on average as compared to 2/329 of the base model, approaching the performance of 4 times larger Qwen3-32B (18.4/329) and surpassing GPT-4o-mini (10/329). To facilitate reproducibility and future research in this direction, we make our code, model weights, and generated scripts publicly available [2].

The rest of the manuscript is organized as follows. We detail the task and motivate the method design in Section 2, describe the training and evaluation approach in Section 3, and provide a comprehensive overview of our experimental results in Section 4.

2 Environment Setup

In this section, we provide the formulation of the environment setup task following previous work and present our initial analysis of the problems faced by general-purpose LLMs.

2.1 Task Definition

Recently, several environment setup benchmarks were proposed [Eliseeva et al., 2025, Hu et al., 2025b, Milliken et al., 2025, Arora et al., 2025]. In this study, we focus on EnvBench [Eliseeva et al., 2025] due to its large scale and lightweight static analysis-based metrics. In total, EnvBench comprises 665 JVM and 329 Python repositories curated from GitHub. We consider Python part of EnvBench and leave exploration on other languages for future work. We follow the original task definition, which we briefly outline below.

[2]Replication Package: `https://github.com/envsetup-rl-dl4c/envsetup-rl`

Input The environment setup approach has access to the full repository context and base environment configuration. How exactly this context is utilized remains part of the approach definition: it could be a predefined prompt, an interactive agentic workflow, and more.

Output The environment setup approach should produce a shell script that successfully installs all the needed dependencies in the base environment.

Evaluation The correctness of the environment setup script is evaluated by first executing it, and then, if the script was executed successfully (finished with exit code 0), invoking Pyright[3]—a static analysis tool used to evaluate whether the imports across the codebase were resolved successfully. The repository is considered to be set up correctly if the script finished with exit code 0 and subsequent Pyright check reported no import issues.

2.2 Exploratory Analysis

As was highlighted by Eliseeva et al. [2025], even frontier models achieve modest results on EnvBench. To shed light on core problems faced by the LLMs during environment setup, we qualitatively study the scripts generated by GPT-4o for a sample of 40 repositories. Overall, we find that failures are due to the inability of the models to fully understand the context of the repository, the system they operate in, and the tools that they are required to use.

Specifically, we identify 11 failure patterns in model-produced scripts and 3 configuration challenges presented by the repositories that GPT-4o could not overcome. These failures fall into two categories: those producing non-zero exit codes, dominated by incorrect syntax (10% of repositories) and models failing to resolve conflicting dependencies versions (7.5%), and those causing unresolved import issues reported by Pyright, most frequently, models failing to install dependencies present in the codebase but not specified in the configuration files (25%) and optional dependencies required for development, such as test packages or linters (22.5%). Finally, we note 7 Script Problems patterns that could potentially be detected simply by localized check driven by parsing, static analysis, and heuristics. A detailed description of the analysis process and all findings are presented in Appendix B.

3 Method

The lack of a strong teacher model complicates creating a high-quality training dataset for traditional supervised learning approaches common in the literature [Pan et al., 2025]. RLVR is a promising alternative that requires not a high-quality supervised dataset but rather a way to verify the correctness of the results. Moreover, RLVR showed benefits in various domains, including SE [Luo et al., 2025, Golubev et al., 2025]. Hence, we apply RLVR to the environment setup task. As illustrated in Figure 1, our RLVR training pipeline samples candidate outputs from a model, scores them with a verifiable reward function to obtain scalar rewards, and updates model parameters via the chosen RLVR algorithm.

The rest of the section is structured as follows. First, in Section 3.1 we introduce our training setup. Then, in Section 3.2 we describe the motivation and implementation of the proposed rewards. Finally, in Section 3.3 we describe how we evaluate the results.

3.1 Training setup

Algorithm. We train our policy with REINFORCE++ [Hu et al., 2025a], a critic-free method that stabilizes updates via global batch advantage normalization and lowers the compute by removing the value model. We do not adopt GRPO [Shao et al., 2024] because it requires generating multiple samples for each prompt and would significantly increase experiment duration in our setting. In practice, REINFORCE++ provides comparable training dynamics and strong generalization with a much lighter footprint. A more exhaustive comparison with GRPO and GRPO-like objectives is left for future work.

Models. We use Qwen3-8B as our base model, selected for its strong performance on SE tasks and reasonable compute requirements [Qwen Team, 2025]. Qwen models also show consistent improvements with RLVR training compared with other model families [Gandhi et al., 2025]. We

[3] `https://microsoft.github.io/pyright`

leave the exploration of other model families and model sizes to future work. We use non-thinking mode because reasoning traces are often long [Sui et al., 2025] and increase the GPU memory requirements as well as the training duration.

Scaffold. Our experiments follow zero-shot approach from Eliseeva et al. [2025]. The model is prompted with the general task description, predefined context for the particular repository, and information about the base environment (Dockerfile contents). It generates a shell script in a single attempt without receiving any intermediate feedback from the environment. We instruct the model to provide a script in a Markdown format, enclosed in ```bash and ``` delimiters. The prompts and the provided repository context are described in Appendix A.1.

Data. We perform both training and evaluation on the Python subset of EnvBench. Following recent work on code benchmarks [Gehring et al., Jain et al., 2025, Le et al., 2022], where agents learn through trial-and-error on the same problems used for evaluation, our setup also employs EnvBench tasks for both training and evaluation. However, we never explicitly provide any ground-truth labels to the model, only rule-based reward scores for the generated scripts. This ensures the model cannot trivially memorize correct answers, forcing it to learn from reward feedback alone. We additionally compare the results on the train and validation sets in Appendix C and find no strong indication of memorization. Specifically, we reserve 96 repositories as a held-out validation set and use the remaining 228 repositories for training. Due to technical issues, we omit five repositories from EnvBench from our training and validation sets. To save the compute, we cache the zero-shot prompt for each repository, instead of dynamically composing it for each training step. The resulting dataset is available online[4].

Framework and Hyperparameters. We use the VeRL framework [Sheng et al., 2024] for training. All our training runs are executed on 4xH200 GPUs. We set a batch size of 64 and the number of epochs to 15, yielding 45 training steps. We truncate the prompts longer than 30000 tokens and allow the model to generate up to 4096 tokens in response. We use vLLM [Kwon et al., 2023] as the rollout engine and set sampling parameters to the values recommended in Qwen3 model card for non-thinking mode. We perform 5 optimization epochs on each trajectory batch to improve sample efficiency. We use AdamW [Loshchilov and Hutter] optimizer. One training run takes 4 hours on average. Our comprehensive hyperparameter setup and training details are listed in Appendix A.2.

3.2 Proxy Rewards

The reward design is a crucial component of RLVR training. A common choice is to use binary outcome-based rewards for each model response [Luo et al., 2025]. For the environment setup task, this means evaluating whether each script successfully configures the corresponding repository. For safety, each script must run in an isolated container, which, together with the massive scale required for efficient RLVR training (e.g., recent work runs up to 512 containers in parallel [Luo et al., 2025]), creates significant computational and technical overhead. To address these challenges, we introduce three lightweight execution-free *proxy rewards* based on our qualitative analysis from Section 2 and experiment with them independently to probe alternative signals. Throughout the following reward definitions, we use o for the model output and $s = \text{extract\_script}(o)$ for the environment-setup shell script it contains. extract_script uses regular expressions to parse the shell script from the model outputs; if parsing fails, we consider s to be empty.

LLM-as-a-Judge This reward (denoted R_{LLM}) takes in the extracted script s along with a comprehensive context for the corresponding repository and emulates the EnvBench evaluation suite. The judge predicts the exit code from the shell script execution and the number of Pyright issues (num_issues). We use GPT-4.1 as the backbone LLM for the judge. Further implementation details could be found in Appendix A.5. The reward is calculated as follows:

$$R_{\text{LLM}}(s) = \begin{cases} -1.0, & \text{if } s \text{ is empty} \\ 0.0, & \text{if exit\_code}(s) \neq 0 \\ \max\left(1.0 - \frac{\text{num\_issues}(s)}{100},\ 0.0\right), & \text{otherwise} \end{cases}$$

[4] https://huggingface.co/envsetup-rl-dl4c

Heuristics This reward ($R_{\text{heuristics}}$) also analyzes both the repository contents and the model-generated script. It parses the repository's configuration files and the commands from the model-generated script, then applies rule-based heuristics to detect two categories of issues: major issues (logical mistakes that lead to a non-zero exit code after script execution) and minor issues (logical mistakes that lead to unresolved import issues after script execution). Based on qualitative analysis of environment setup fault patterns described in Appendix B, we develop 4 major and 1 minor issue types, with details in Appendix A.6. The reward is calculated as follows:

$$R_{\text{heuristics}}(s) = \begin{cases} -1.0, & \text{if } s \text{ is empty} \\ -0.5, & \text{if major issues are observed in } s \\ 0.5, & \text{if minor issues are observed in } s \\ 1.0, & \text{if no issues are observed in } s \end{cases}$$

ShellCheck This reward ($R_{\text{ShellCheck}}$) relies on ShellCheck[5], a popular static analysis tool for shell scripts. Given a shell script, ShellCheck outputs a list of the observed issues, including syntax errors, stylistic problems, and more[6]. Similar to how the feedback from static analysis tools for code is helpful for code generation agents [Jiang et al., 2025], we hypothesize that ShellCheck could improve the shell script generation quality for our end task, environment setup. The reward is calculated based on a binary check that penalizes for any issues reported by ShellCheck, as follows:

$$R_{\text{ShellCheck}}(s) = \begin{cases} -1.0, & \text{if } s \text{ is empthy} \\ 0.0, & \text{if ShellCheck reports any issues for } s \\ 1.0, & \text{otherwise} \end{cases}$$

3.3 Evaluation Setup

Baselines. We consider multiple general-purpose LLMs in the same zero-shot scaffold that we use for our experiments. Following Eliseeva et al. [2025], we compare our results with two closed-source models from OpenAI, GPT-4o and GPT-4o-mini. Additionally, we consider multiple models from the Qwen3 family (specifically, 8B, 14B, and 32B). All models are used in non-thinking mode.

Data. We conduct the evaluation on the full set of 329 Python datapoints from EnvBench.

Metrics. We build off the metrics proposed in EnvBench and consider pass@k—the binary measure of success across k attempts for each datapoint. It is equal to 1 for a given repository, if at least once in k attempts the model was able to generate a script that results in an exit code of 0 and no issues reported by Pyright. Another metric we introduce for more detailed results analysis is *avgFixRate*—the percentage of Pyright issues resolved by running the generated script averaged across all repositories. This metric is equal to 100% for the successfully installed repositories, and to 0% for the repositories with a non-zero exit code. We cache the number of Pyright issues before running the script to lower computational costs. We also report *# Failed*—number of repositories where the scripts resulted in a non-zero exit code.

Unlike Eliseeva et al. [2025], we conduct five independent evaluation runs for each model, since we observe high variance in the performance of trained models. We separately report pass@5 and all other metrics, averaged across the five runs.

4 Experiments

In this section, we provide and discuss the achieved experimental results following the methodology outlined in Section 3. We first discuss the dynamics of the training, and then the evaluation results.

4.1 Training Dynamics

Training dynamics with each proxy reward described in Section 3.2 are depicted in Figure 2. Every reward function returns values from the $[-1, 1]$ range, where -1 indicates malformed scripts, and 1 indicates perfect performance.

[5] `https://www.shellcheck.net`

[6] The full list of issues detectable by ShellCheck is available online: `https://www.shellcheck.net/wiki/`

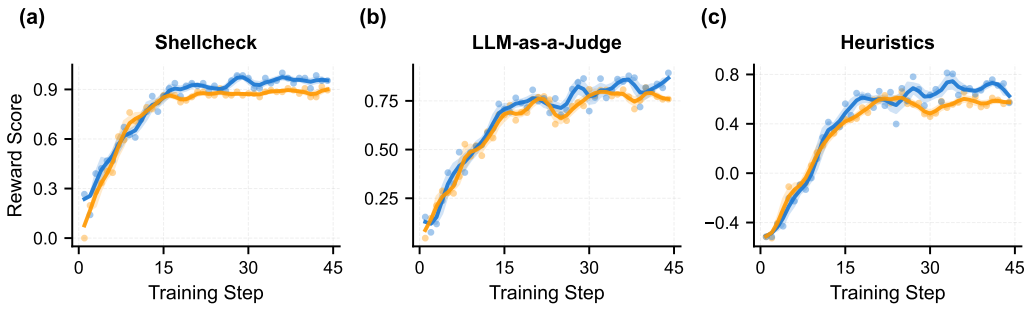

Figure 2: RLVR training dynamics with each of the proxy rewards described in Section 3.2. Raw datapoints are shown as semi-transparent dots, with Gaussian-smoothed curves overlaid to highlight trends. **Blue** shows average reward on the training set; **orange** shows average reward on the validation set. The x-axis is training steps, and the y-axis is average reward. **(a)** Evolution of the ShellCheck reward $R_{\text{ShellCheck}}$, **(b)** LLM-as-a-Judge reward R_{LLM}, and **(c)** Heuristics reward $R_{\text{heuristics}}$.

The base model exhibits formatting compliance but fails to satisfy the meaningful criteria imposed by each reward. This is evident from **validation** scores at step 0, which are close to the minimum values achievable with the correct formatting: near 0.0 for LLM-as-a-Judge and ShellCheck, and near -0.5 for Heuristics reward.

Across all rewards, we observe a steady initial increase for both **training** and **validation** sets, which then slows for LLM-as-a-Judge and Heuristics rewards, and largely plateaus for ShellCheck. The substantial differences in validation reward scores between step 0 and step 45 suggest that RLVR training successfully steers the model to better adhere to the criteria imposed by each reward. In addition, we do not observe strong overfitting: there is only a small gap between **training** and **validation** reward scores.

ShellCheck approaches saturation with a mean **validation** score of 0.9 at step 45, while LLM-as-a-Judge and Heuristics rewards achieve lower scores of 0.76 and 0.58, respectively. By definition, ShellCheck represents the most straightforward reward in our set, as achieving a perfect score requires only passing general script quality checks. The lower performance of LLM-as-a-Judge and Heuristics rewards likely reflects Qwen3-8B's difficulty satisfying their more challenging task-specific criteria. We leave the hyperparameter tuning to improve the training dynamics for future work.

4.2 Evaluation Results

The evaluation results on EnvBench are presented in Table 1. The best model selection strongly depends on the metrics priority, highlighting the complexity of the benchmark. For example, while Qwen3-32B achieves the highest pass@5 (33/329), GPT-4o demonstrates the highest per-run performance with an average pass@1 of 21.2 ± 1.5 across five runs.

The best of the proposed methods, LLM-as-a-Judge, not only significantly outperforms its base model across all metrics, but also shows the best avgFixRate across all tested models. It also surpasses the pass@5 (22) of larger models Qwen3-14B (17) and GPT-4o-mini (16), and gets the avg pass@1 (15.8 ± 1.3) close to that of Qwen3-32B (18.4 ± 1.8), the model we consider our upper bound. Figure 3 compares the LLM-as-a-Judge checkpoint with all baselines, highlighting how our approach substantially improves the performance-cost balance for Qwen3-8B.

Beyond the increase in fully configured repositories, RLVR-tuned models show progress in partial configuration through decreased # Failed and increased avgFixRate, suggesting better understanding of the task. For example, while Heuristics-tuned Qwen3-8B is near the level of the base Qwen3-14B model in terms of pass@5 and averaged pass@1, it produces substantially fewer failing scripts (222 ± 7 vs. 260 ± 7) and resolves more import errors ($(17 \pm 2)\%$ vs. $(11.5 \pm 1.8)\%$).

Since the best-performing LLM-as-a-Judge relies on a powerful GPT-4.1 model to robustly provide targeted feedback and carries associated costs, each training run required approximately $250 in API costs alone. Heuristics reward offers a compelling alternative, providing smaller but substantial improvements (e.g., pass@5 of 17 vs. 22 for LLM-as-a-Judge) via lightweight static analysis checks

Table 1: EnvBench evaluation results for base models and our RLVR-tuned Qwen3-8B with various rewards. Total number of samples is 329. **pass@5** shows the number of successful samples (zero exit code and zero issues). **avg@5** shows mean ± std for the following metrics: **# Success** (average number of successful samples per run), **# Failed** (average number of samples where scripts finished with non-zero exit code), and **avgFixRate** (average ratio of resolved import issues per sample as compared to the evaluation run with empty setup script; for samples where scripts execute with non-zero exit codes, ratio is considered 0). The symbol ↑ indicates higher is better, while ↓ indicates lower is better.

Model	Reward	pass@5	avg@5		
		# Success ↑	# Success ↑	# Failed ↓	avgFixRate ↑
Proprietary Baselines					
GPT-4o	—	31	$\mathbf{21.2 \pm 1.5}$	185 ± 3	$(28 \pm 4)\%$
GPT-4o-mini	—	16	10.0 ± 1.6	$\mathbf{160 \pm 4}$	$(24.1 \pm 1.4)\%$
Open-Weight Baselines					
Qwen3-32B	—	**33**	18.4 ± 1.8	190 ± 5	$(28.0 \pm 1.4)\%$
Qwen3-14B	—	17	7 ± 2	260 ± 7	$(11.5 \pm 1.8)\%$
Qwen3-8B	—	6	2.0 ± 0.7	289 ± 3	$(5.1 \pm 0.8)\%$
RLVR-tuned (ours)					
Qwen3-8B	LLM-as-a-Judge	22	15.8 ± 1.3	182 ± 2	$\mathbf{(30.7 \pm 0.5)\%}$
	Heuristics	17	6 ± 2	222 ± 7	$(17 \pm 2)\%$
	ShellCheck	7	2.8 ± 1.3	281 ± 4	$(6.9 \pm 0.6)\%$

alone. When employing larger models and, therefore, longer training, it may serve as a viable option to save on compute.

The ShellCheck reward serves as an ablation study underlining the conclusions from the Section 2.2. Its performance, while still marginally over the base model, is far from the other rewards, while the rewards both on the train and validation sets are high. Analysis of pre- and post-training scripts reveals that ShellCheck-identifiable issues do not address the core problems causing script execution failures or partial setup. Common remaining issues include environment-specific configuration errors and logical mistakes in setup sequences that pass ShellCheck checks but fail in practice. More details can be found in Appendix B. For transparency, in addition to our main results, in Appendix D.1 we note a very strong ShellCheck checkpoint result that we were unable to reproduce.

Finally, we verify the generalization of our RLVR-tuned models in Appendix C. We confirm that the conclusions presented in this section hold when evaluated on the held-out validation set alone and observe no strong signs of memorization.

5 Related Work

Environment Setup. Following the advances of LLMs in other SE tasks [Liu et al., 2024], previous works extensively explored their applications to the environment setup task. Several environment setup benchmarks were introduced, such as EnvBench [Eliseeva et al., 2025], Repo2Run [Hu et al., 2025b], and others [Milliken et al., 2025, Arora et al., 2025]. They differ in scale (from tens to hundreds of repositories), expected model outputs (shell scripts or Dockerfiles), and metrics (static analysis or test-based). Our study required a large sample of Python repositories, which left us with EnvBench (329 repositories) and Repo2Run (420 repositories). We selected EnvBench because its static analysis-based metrics are more lightweight than Repo2Run's test-based metrics and facilitate faster experiments.

Existing environment setup approaches range from simple zero-shot prompts [Badertdinov et al., 2025, Eliseeva et al., 2025, Li et al., 2025] to complicated agentic workflows [Milliken et al., 2025, Bouzenia and Pradel, 2025, Hu et al., 2025b, Vergopoulos et al., Zhang et al., 2025, Guo et al., 2025]. Existing works use general-purpose LLMs as backbones, and many workflows include execution of

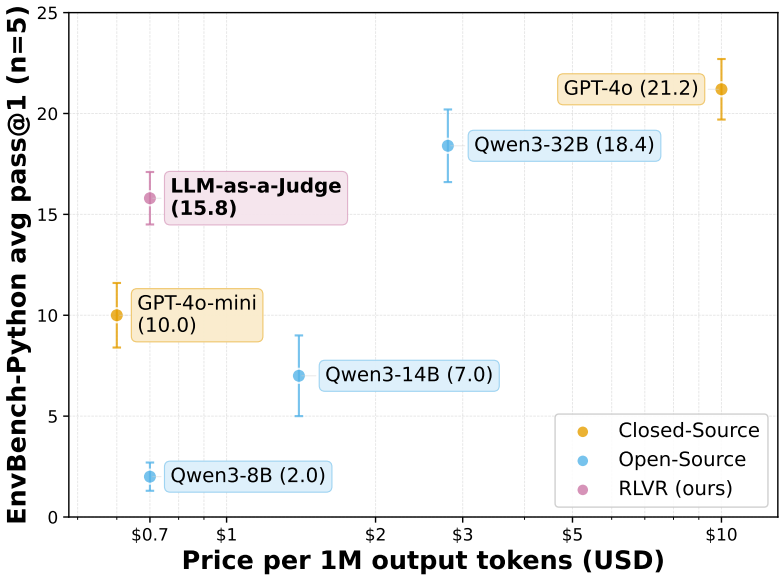

Figure 3: Performance on EnvBench-Python (pass@1 averaged across five runs) compared to the costs of running models. To exceed the performance of the model yielded by the proposed method, one needs to quadruple the costs. The prices for Qwen3 models are taken from the Alibaba Cloud website: `https://www.alibabacloud.com/help/en/model-studio/models`.

intermediate agent outputs [Eliseeva et al., 2025, Milliken et al., 2025, Bouzenia and Pradel, 2025, Hu et al., 2025b, Vergopoulos et al., Zhang et al., 2025, Guo et al., 2025], introducing isolation and cost considerations. In contrast, we focus on a zero-shot scaffold, which was previously shown to achieve reasonable performance given its simplicity [Eliseeva et al., 2025, Badertdinov et al., 2025], to study how far LLMs can go under consistent constraints. Finally, we note that many works use automated environment setup approaches as a mere tool for constructing SWE-bench-like [Jimenez et al., 2024] datasets [Badertdinov et al., 2025, Vergopoulos et al., Zhang et al., 2025, Guo et al., 2025], making the environment setup not the primary research focus.

Reinforcement Learning with Verifiable Rewards (RLVR). Reinforcement Learning (RL) has emerged as a powerful LLM post-training technique to further enhance the model's capabilities, with early successes achieved from human feedback [Christiano et al., 2017, Kaufmann et al., 2024]. Building on this foundation, the RLVR has gained traction, wherein the reward signal is provided by a rule-based or programmatic verifier. RLVR has found particularly impactful applications in domains such as mathematics [Lambert et al., 2025, Feng et al., 2025] and code generation [Wei et al., 2025, Luo et al., 2025, Golubev et al., 2025].

The effectiveness of RLVR has been amplified by recent advances in RL algorithms building upon Proximal Policy Optimization (PPO) [Schulman et al., 2017] (e.g., VAPO [Yue et al., 2025], RLOO [Kool et al., 2019, Ahmadian et al., 2024], Reinforce++ [Hu et al., 2025a], GRPO [Shao et al., 2024], DAPO [Yu et al., 2025], Dr. GRPO [Liu et al., 2025], GRPO++ [Luo et al., 2025], GSPO [Zheng et al., 2025]). Furthermore, recent research has explored RLVR settings that do not rely on labeled data [Zhao et al., 2025] or even operate without an explicit verifier [Zhou et al., 2025].

6 Limitations and Future Work

In this section, we outline key limitations of our study and promising directions for future work.

Models We apply the proposed framework to a single LLM, Qwen3-8B in non-thinking mode. While it comes from the widely used Qwen3 family and presents a competitive quality-compute tradeoff, the range of applicability of our study could be further verified by probing other model families, different model sizes, and reasoning LLMs.

Scaffold We consider a simple single-turn scaffold in our experiments. Previous works on environment setup suggest that multi-turn agentic scaffolds—which iteratively interact with an environment and refine their solutions based on the feedback received on each step—could bring significant improvements. Extending RLVR training to such multi-turn scaffolds represents a natural progression for enhancing environment setup capabilities.

Proxy Rewards We introduce lightweight reward functions that allow for the RLVR training pipeline without computational overhead on scaling containerized execution. While we consider this direction promising given its light computation burden and obtained results, ground truth runtime feedback would likely provide richer training signals and drive further performance gains.

Generalization We use EnvBench for both training and evaluation. While we do not provide any ground truth labels or feedback during training and observe no strong signs of memorization (Appendix C), incorporating additional environment setup benchmarks would better establish the generalization capabilities of our trained checkpoints.

7 Conclusion

Our study demonstrates that online Reinforcement Learning with Verifiable Rewards (RLVR) can significantly improve environment setup capabilities in open-source models. To avoid computationally expensive ground truth evaluation, we design three lightweight execution-free proxy rewards—LLM-as-a-Judge evaluation, static analysis with ShellCheck, and heuristics-based verification tailored to the environment setup task—all of which yield improvements over the performance of the base model, Qwen3-8B. In particular, our best-performing checkpoint, Qwen3-8B with LLM-as-a-Judge reward, is able to set up 15.8 out of 329 repositories in EnvBench-Python on average over five runs. To move beyond this performance without fine-tuning, a 4 times larger Qwen3-32B model is needed. The trained model also fixes more imports than any of the investigated baselines. Our replication package with training code and trained model checkpoints is available online: `https://github.com/envsetup-rl-dl4c/envsetup-rl`.

References

Arash Ahmadian, Chris Cremer, Matthias Gallé, Marzieh Fadaee, Julia Kreutzer, Olivier Pietquin, Ahmet Üstün, and Sara Hooker. Back to basics: Revisiting reinforce style optimization for learning from human feedback in llms, 2024. URL `https://arxiv.org/abs/2402.14740`.

Avi Arora, Jinu Jang, and Roshanak Zilouchian Moghaddam. Setupbench: Assessing software engineering agents' ability to bootstrap development environments. *arXiv preprint arXiv:2507.09063*, 2025.

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

Islem Bouzenia and Michael Pradel. You name it, i run it: An llm agent to execute tests of arbitrary projects. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1054–1076, 2025.

Jianming Chang, Xin Zhou, Lulu Wang, David Lo, and Bixin Li. Bridging bug localization and issue fixing: A hierarchical localization framework leveraging large language models, 2025. URL `https://arxiv.org/abs/2502.15292`.

Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. LocAgent: Graph-guided LLM agents for code localization. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar, editors, *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 8697–8727, Vienna, Austria, July 2025. Association for Computational Linguistics. ISBN 979-8-89176-251-0. doi: 10.18653/v1/2025.acl-long.426. URL `https://aclanthology.org/2025.acl-long.426/`.

Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. *Advances in neural information processing systems*, 30, 2017.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

Aleksandra Eliseeva, Alexander Kovrigin, Ilia Kholkin, Egor Bogomolov, and Yaroslav Zharov. Envbench: A benchmark for automated environment setup, 2025. URL https://arxiv.org/abs/2503.14443.

Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjun Zhong. Retool: Reinforcement learning for strategic tool use in llms, 2025. URL https://arxiv.org/abs/2504.11536.

Kanishk Gandhi, Ayush Chakravarthy, Anikait Singh, Nathan Lile, and Noah D Goodman. Cognitive behaviors that enable self-improving reasoners, or, four habits of highly effective stars. *arXiv preprint arXiv:2503.01307*, 2025.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. In *Forty-second International Conference on Machine Learning*.

Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, Sergei Skvortsov, and Boris Yangel. Training long-context, multi-turn software engineering agents with reinforcement learning, 2025. URL https://arxiv.org/abs/2508.03501.

Lianghong Guo, Yanlin Wang, Caihua Li, Pengyu Yang, Jiachi Chen, Wei Tao, Yingtian Zou, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*, 2025.

Md Mahade Hasan, Muhammad Waseem, Kai-Kristian Kemell, Jussi Raskua, Juha Ala-Rantalaa, and Pekka Abrahamsson. Assessing small language models for code generation: An empirical study with benchmarks. *arXiv preprint arXiv:2507.03160*, 2025.

Jian Hu, Jason Klein Liu, Haotian Xu, and Wei Shen. Reinforce++: An efficient rlhf algorithm with robustness to both prompt and reward models, 2025a. URL https://arxiv.org/abs/2501.03262.

Ruida Hu, Chao Peng, Xinchen Wang, Junjielong Xu, and Cuiyun Gao. Repo2run: Automated building executable environment for code repository at scale, 2025b. URL https://arxiv.org/abs/2502.13681.

idank. bashlex: Python parser for bash. GitHub repository, 2025. URL https://github.com/idank/bashlex. Accessed: 2025-08-18.

Arnav Kumar Jain, Gonzalo Gonzalez-Pumariega, Wayne Chen, Alexander M Rush, Wenting Zhao, and Sanjiban Choudhury. Multi-turn code generation through single-step rewards. *arXiv preprint arXiv:2502.20380*, 2025.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *The Thirteenth International Conference on Learning Representations*.

Hao Jiang, Qi Liu, Rui Li, Yuze Zhao, Yixiao Ma, Shengyu Ye, Junyu Lu, and Yu Su. Verse: Verification-based self-play for code instructions. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 24276–24284, 2025.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Timo Kaufmann, Paul Weng, Viktor Bengs, and Eyke Hüllermeier. A survey of reinforcement learning from human feedback, 2024. URL https://arxiv.org/abs/2312.14925.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, Zhuofu Chen, Jialei Cui, Hao Ding, Mengnan Dong, Angang Du, Chenzhuang Du, Dikang Du, Yulun Du, Yu Fan, Yichen Feng, Kelin Fu, Bofei Gao, Hongcheng Gao, Peizhong Gao, Tong Gao, Xinran Gu, Longyu Guan, Haiqing Guo, Jianhang Guo, Hao Hu, Xiaoru Hao, Tianhong He, Weiran He, Wenyang He, Chao Hong, Yangyang Hu, Zhenxing Hu, Weixiao Huang, Zhiqi Huang, Zihao Huang, Tao Jiang, Zhejun Jiang, Xinyi Jin, Yongsheng Kang, Guokun Lai, Cheng Li, Fang Li, Haoyang Li, Ming Li, Wentao Li, Yanhao Li, Yiwei Li, Zhaowei Li, Zheming Li, Hongzhan Lin, Xiaohan Lin, Zongyu Lin, Chengyin Liu, Chenyu Liu, Hongzhang Liu, Jingyuan Liu, Junqi Liu, Liang Liu, Shaowei Liu, T. Y. Liu, Tianwei Liu, Weizhou Liu, Yangyang Liu, Yibo Liu, Yiping Liu, Yue Liu, Zhengying Liu, Enzhe Lu, Lijun Lu, Shengling Ma, Xinyu Ma, Yingwei Ma, Shaoguang Mao, Jie Mei, Xin Men, Yibo Miao, Siyuan Pan, Yebo Peng, Ruoyu Qin, Bowen Qu, Zeyu Shang, Lidong Shi, Shengyuan Shi, Feifan Song, Jianlin Su, Zhengyuan Su, Xinjie Sun, Flood Sung, Heyi Tang, Jiawen Tao, Qifeng Teng, Chensi Wang, Dinglu Wang, Feng Wang, Haiming Wang, Jianzhou Wang, Jiaxing Wang, Jinhong Wang, Shengjie Wang, Shuyi Wang, Yao Wang, Yejie Wang, Yiqin Wang, Yuxin Wang, Yuzhi Wang, Zhaoji Wang, Zhengtao Wang, Zhexu Wang, Chu Wei, Qianqian Wei, Wenhao Wu, Xingzhe Wu, Yuxin Wu, Chenjun Xiao, Xiaotong Xie, Weimin Xiong, Boyu Xu, Jing Xu, Jinjing Xu, L. H. Xu, Lin Xu, Suting Xu, Weixin Xu, Xinran Xu, Yangchuan Xu, Ziyao Xu, Junjie Yan, Yuzi Yan, Xiaofei Yang, Ying Yang, Zhen Yang, Zhilin Yang, Zonghan Yang, Haotian Yao, Xingcheng Yao, Wenjie Ye, Zhuorui Ye, Bohong Yin, Longhui Yu, Enming Yuan, Hongbang Yuan, Mengjie Yuan, Haobing Zhan, Dehao Zhang, Hao Zhang, Wanlu Zhang, Xiaobin Zhang, Yangkun Zhang, Yizhi Zhang, Yongting Zhang, Yu Zhang, Yutao Zhang, Yutong Zhang, Zheng Zhang, Haotian Zhao, Yikai Zhao, Huabin Zheng, Shaojie Zheng, Jianren Zhou, Xinyu Zhou, Zaida Zhou, Zhen Zhu, Weiyu Zhuang, and Xinxing Zu. Kimi k2: Open agentic intelligence, 2025. URL https://arxiv.org/abs/2507.20534.

Wouter Kool, Herke van Hoof, and Max Welling. Buy 4 REINFORCE samples, get a baseline for free!, 2019. URL https://openreview.net/forum?id=r1lgTGL5DE.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tülu 3: Pushing frontiers in open language model post-training. 2024.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, Yuling Gu, Saumya Malik, Victoria Graf, Jena D. Hwang, Jiangjiang Yang, Ronan Le Bras, Oyvind Tafjord, Chris Wilhelm, Luca Soldaini, Noah A. Smith, Yizhong Wang, Pradeep Dasigi, and Hannaneh Hajishirzi. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL https://arxiv.org/abs/2411.15124.

langchain-ai. langgraph: Build resilient language agents as graphs. GitHub repository, 2025. URL https://github.com/langchain-ai/langgraph. Accessed: 2025-08-18.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems*, 35:21314–21328, 2022.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting large language models to tackle the full software development lifecycle: A case study. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert, editors, *Proceedings of the 31st International Conference on Computational*

Linguistics, pages 7511–7531, Abu Dhabi, UAE, January 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.coling-main.502/.

Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey, 2024. URL https://arxiv.org/abs/2409.02977.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding r1-zero-like training: A critical perspective, 2025. URL https://arxiv.org/abs/2503.20783.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*.

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

Zexiong Ma, Chao Peng, Qunhong Zeng, Pengfei Gao, Yanzhen Zou, and Bing Xie. Tool-integrated reinforcement learning for repo deep search, 2025. URL https://arxiv.org/abs/2508.03012.

Louis Milliken, Sungmin Kang, and Shin Yoo. Beyond pip install: Evaluating llm agents for the automated installation of python projects. In *2025 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1–11. IEEE Computer Society, 2025.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2025. URL https://arxiv.org/abs/2412.21139.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

Qwen Team. Qwen3 technical report, 2025. URL https://arxiv.org/abs/2505.09388.

Revanth Gangi Reddy, Tarun Suresh, JaeHyeok Doo, Ye Liu, Xuan Phi Nguyen, Yingbo Zhou, Semih Yavuz, Caiming Xiong, Heng Ji, and Shafiq Joty. Swerank: Software issue localization with code ranking. *arXiv preprint arXiv:2505.07849*, 2025.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017. URL https://arxiv.org/abs/1707.06347.

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, 2024. URL https://arxiv.org/abs/2402.03300.

Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv: 2409.19256*, 2024.

Yang Sui, Yu-Neng Chuang, Guanchu Wang, Jiamu Zhang, Tianyi Zhang, Jiayi Yuan, Hongyi Liu, Andrew Wen, Shaochen Zhong, Hanjie Chen, et al. Stop overthinking: A survey on efficient reasoning for large language models. *arXiv preprint arXiv:2503.16419*, 2025.

Konstantinos Vergopoulos, Mark Niklas Mueller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. In *Forty-second International Conference on Machine Learning*.

Xingyao Wang. Introducing OpenHands LM-32B — a strong, open coding agent model. All-Hands.dev blog, Mar 2025. https://www.all-hands.dev/blog/introducing-openhands-lm-32b----a-strong-open-coding-agent-model (accessed August 12, 2025).

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, 2025. URL https://arxiv.org/abs/2502.18449.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Mu Qiao, Yonghui Wu, and Mingxuan Wang. Dapo: An open-source llm reinforcement learning system at scale, 2025. URL https://arxiv.org/abs/2503.14476.

Yu Yue, Yufeng Yuan, Qiying Yu, Xiaochen Zuo, Ruofei Zhu, Wenyuan Xu, Jiaze Chen, Chengyi Wang, TianTian Fan, Zhengyin Du, Xiangpeng Wei, Xiangyu Yu, Gaohong Liu, Juncai Liu, Lingjun Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Chi Zhang, Mofan Zhang, Wang Zhang, Hang Zhu, Ru Zhang, Xin Liu, Mingxuan Wang, Yonghui Wu, and Lin Yan. Vapo: Efficient and reliable reinforcement learning for advanced reasoning tasks, 2025. URL https://arxiv.org/abs/2504.05118.

Liang Zeng, Yongcong Li, Yuzhen Xiao, Changshi Li, Chris Yuhao Liu, Rui Yan, Tianwen Wei, Jujie He, Xuchen Song, Yang Liu, and Yahui Zhou. Skywork-swe: Unveiling data scaling laws for software engineering in llms, 2025. URL https://arxiv.org/abs/2506.19290.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

Andrew Zhao, Yiran Wu, Yang Yue, Tong Wu, Quentin Xu, Yang Yue, Matthieu Lin, Shenzhi Wang, Qingyun Wu, Zilong Zheng, and Gao Huang. Absolute zero: Reinforced self-play reasoning with zero data, 2025. URL https://arxiv.org/abs/2505.03335.

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, 2025. URL https://arxiv.org/abs/2507.18071.

Xiangxin Zhou, Zichen Liu, Anya Sims, Haonan Wang, Tianyu Pang, Chongxuan Li, Liang Wang, Min Lin, and Chao Du. Reinforcing general reasoning without verifiers, 2025. URL https://arxiv.org/abs/2505.21493.

A Implementation Details

In this section, we provide additional details on our experiments.

A.1 Scaffold Details

We use the same zero-shot scaffold as in Eliseeva et al. [2025]. The prompt is provided in Figure 4. We collect the repository context by running the following bash commands:

```
tree -a -L 3 --filelimit 100 || ls -R
for f in README.md INSTALL.md SETUP.md docs/INSTALL.md docs/SETUP.md; do
  if [ -f "$f" ]; then echo -e "\n=== $f ==="; cat "$f"; fi
done
find . -type f \( \
  -name "*requirements*.txt" -o -name "setup.py" -o -name "pyproject.toml" -o -name "
      setup.cfg" -o -name "tox.ini" \
\) | while read f; do echo -e "\n=== $f ==="; cat "$f"; done
find . -type f -name "*.py" -exec grep -l "python_version\|python_requires" {} \;
find . -type f \( -name ".env*" -o -name "*.env" -o -name "Dockerfile*" \) | \
while read f; do echo -e "\n=== $f ==="; cat "$f"; done
```

Zero-shot Prompt Overview

System Message:
Your task is to generate a bash script that will set up a Python development environment for a repository mounted in the current directory.
You will be provided with repository context. Follow the build instructions to generate the script.

A very universal script might look like this:

```
{baseline_script}
```

However, your job is to make a script more tailored to the repository context.
It will be only run on a single repository mounted in the current directory that you have information about.
The script must not be universal but setup the environment just for this repository.
Avoid using universal if-else statements and try to make the script as specific as possible.

The script should:

- Install the correct Python version based on repository requirements
- Install all project dependencies from `requirements.txt`, `setup.py`, or `pyproject.toml`
- Install any required system packages

For reference, the script will run in this Docker environment, so most of the tools you need will be available:

```
{dockerfile}
```

IMPORTANT:

- Generate ONLY a bash script – you cannot interact with the system
- The script must be non-interactive (use `-y` flags where needed)
- Base all decisions on the provided repository context. Follow the context instructions.
- Do not use `sudo` – the script will run as root
- If you use `pyenv install`, please use the `-f` flag to force the installation. For example: `pyenv install -f $PYTHON_VERSION`
- The script must be enclosed in ```` ```bash``` ```` code blocks

User Message:
Repository Context:
`context`
Generate a complete bash script that will set up this Python environment.
The script must be enclosed in ```` ```bash``` ```` code blocks, it can rely on the tools available in the Docker environment.

Figure 4: Prompt for the zero-shot scaffold for the environment setup task from Eliseeva et al. [2025]. Baseline script and Dockerfile are the same as theirs. Repository context is collected by executing a fixed set of commands within the repository in the target Docker environment.

A.2 Training Details

We show the hyperparameters used in our training in Table 2. We use learning rate of 5×10^{-6} for Shellcheck reward and LLM-as-a-Judge reward, and 3×10^{-6} for Heuristic-based reward. Sampling parameters are set to the values recommended in the Qwen3 model card[7] for non-thinking mode. Full configuration files and code are available in the reproduction package.

Table 2: Hyperparameters used during training.

Parameter	Value
Model Configuration	
Max Prompt Length	30,000
Max Response Length	4,096
Training Settings	
Train Batch Size	64
Mini-Batch Size	32
Micro-Batch Size	1
Optimizer	AdamW
Gradient Clipping	1.0
Total Steps	45
RL Settings	
Algorithm	Reinforce++ [Hu et al., 2025a]
KL Loss	False
KL Reward	False
Entropy Coefficient	0.001
PPO Epochs	5
N Rollouts	1
Rollout Temperature	0.7
Rollout Top-P	0.8
Rollout Top-K	20

A.3 Evaluation Details

We build off the original implementation provided by EnvBench authors. For Qwen3 models, we set the sampling parameters to the values recommended in the corresponding model cards, same as for training (Appendix A.2). The resulting evaluation suite is available in our replication package.

A.4 Shellcheck Reward Implementation

For each model-generated script, we extract the shell code block using regular expressions. The extracted script is saved to a temporary file and analyzed by running ShellCheck via the command-line interface. We use the command:

```
shellcheck -s bash -f json
```

The ShellCheck output is parsed programmatically: if any issues are reported, the reward is set to 0; if the script is empty, the reward is -1; otherwise, the reward is 1. The implementation is available in our replication package.

A.5 LLM-as-a-Judge Reward Implementation

The LLM-as-a-Judge reward provides repository-specific, scalable feedback for environment setup scripts by using an LLM as an evaluator. The LLM is prompted to simulate the execution of a candidate shell script in EnvBench Docker environment and predict the outcome of the environment setup process, including the script's exit code and the number of missing import issues (as would be detected by Pyright static analysis).

[7] https://huggingface.co/Qwen/Qwen3-8B#best-practices

The prompt provided to the LLM includes the following components: the Dockerfile specifying the environment, evaluation guidelines informed by our exploratory analysis of model-generated scripts, and several few-shot examples illustrating script grading. Complete prompt templates and reward implementation code are available in our replication package.

We selected GPT-4.1 as the language model for our experiments, as it consistently yielded the most reliable results. While we also evaluated GPT-4o and GPT-4o-mini, these models did not achieve comparable performance. In addition, we explored several ablations: (1) augmenting the LLM-as-a-Judge with repository information like the zeroshot context, and (2) replacing the LLM-as-a-Judge with an LLM Agent equipped with tools for repository exploration. However, neither approach led to a significant improvement in model performance. Consequently, we adopted the simplest and most robust configuration for our main experiments.

A.6 Heuristics Reward Implementation

We build the Heuristics reward based on the checklist constructed during the qualitative analysis (refer to Appendix B for further details). We implement a deterministic algorithm that parses both repository contents and the model-generated scripts to answer a subset of questions from the checklist that we found plausible to judge based on simple heuristics. We use LangGraph [langchain-ai, 2025] for implementation, bashlex [idank, 2025] for parsing shell scripts (with fallbacks to regular expressions), tomllib for parsing TOML configuration files, and ast for parsing setup.py configuration file. The implementation is available in our replication package: `https://github.com/envsetup-rl-dl4c/envsetup-rl`.

Specifically, it gathers the following information.

- Repository Contents
  - `configuration_files`: Paths to the configuration files detected in the repository (determined by matching a list of files to the naming conventions for pip and Poetry).
  - `true_dep_manager`: The dependency manager used in the repository (determined based on configuration files naming conventions).
  - `true_dep_groups`: The optional dependency groups specified in the repository configuration files (determined by parsing the configuration files).
  - `true_extras`: The extras specified in the repository configuration files (determined by parsing the configuration files).
  - `python_version_reqs`: Requirements for Python version specified in the repository configuration files (determined by parsing the configuration files).
- Script
  - `used_dep_manager`: The dependency manager(s) invoked by the script (determined by parsing commands from the script).
  - `used_dep_groups`: The optional dependency groups installed by the script (determined by parsing `poetry install` invocation from the script, if present — only relevant for Poetry).
  - `used_extras`: The extras installed by the script (determined by parsing `poetry install` or `pip install` invocations from the script, if present).
  - `installed_python_versions`: Which Python versions are installed by the script (determined by parsing `pyenv install` command invocation from the script, if present).
  - `install_command_validation`: Validation for the dependency installation commands.
    - `has_pip_install`: `bool` — Whether any `pip install` command is present.
    - `has_poetry_install`: `bool` — Whether any `poetry install` command is present.
    - `pip_install_valid`: `bool` — Syntactic validity of `pip install` commands against documented flags; also checks `-r/--requirement` file existence and basic content.
    - `poetry_install_valid`: `bool` — Syntactic validity of `poetry install` commands against documented flags.

    - `pip_validation_issues`: `List[str]` — Issues detected in `pip install` commands, if present.
    - `poetry_validation_issues`: `List[str]` — Issues detected in `poetry install` commands, if present.
- System Configuration
  - `installed_python_versions`: Which Python versions are installed on the system (determined by invoking `python -version` and `pyenv versions`).

The gathered information is used to identify major issues (logical mistakes that lead to a non-zero exit code after script execution) and minor issues (logical mistakes that lead to unresolved import issues after script execution). We identify 4 major issues and 1 minor issue based on the qualitative analysis (Section 2.2, Appendix B) and further detail them as follows.

- Major Issues
  - **Multiple Dep. Managers**, i.e., if `used_dep_manager` contains more than one dependency manager. Using multiple dependency managers within a single setup script is likely to lead to failure.
  - **Wrong Dep. Manager**, i.e., if there is a mismatch between `used_dep_manager` and `true_dep_manager`. For example, if the repository uses pip but the script uses Poetry commands, it will likely lead to failure.
  - **Wrong Syntax**, i.e., either `has_pip_install` and not `pip_install_valid` or `has_poetry_install` and not `poetry_install_valid`. Syntax errors result in a runtime failure.
  - **Non-existent Dep. Groups / Extras**, i.e., `used_dep_groups` contains groups not present in `true_dep_groups` or `used_extras` contains extras not present in `true_extras`. This results in a runtime failure.
- Minor Issues
  - **Missing Dep. Group / Extras**, i.e., `true_dep_groups` is non-empty, and `used_dep_groups` is a subset of `true_dep_groups` or `true_extras` is non-empty, and `used_extras` is a subset of `true_extras`. While the script may execute successfully, some optional dependencies required for full functionality may be missing, resulting in unresolved import errors.

B Empirical Study of Environment Setup Failure Patterns

B.1 Failure Patterns

We manually analyzed scripts generated by GPT-4o in a zero-shot scaffold, the second-best approach on EnvBench, to understand the fault modes of environment setup scripts. Specifically, we selected 40 scripts (from the first 40 repositories in lexicographical order where the results were available). Out of those repositories, 2 were set up correctly, 16 had a non-zero exit code (failed), and 22 had unresolved import issues. For each script, we collected free-form observations about potential failure reasons and applied an open coding approach to extract common failure themes.

We present the resulting failure patterns in Table 3, with labels for 40 repositories available in our replication package. We identify three failure patterns categories: (i) Script Problems, the explicit mistakes made in the model-generated scripts, (ii) Repository Problems, the configuration challenges presented by a specific repository that the model failed to consider, and (iii) Eval Problems, runtime failures of EnvBench evaluation suite and/or limitations of the static analysis. Most failures are caused by Script Problems, while unresolved import issues are often due to Repository Problems. We observe 3 Eval Problems in total (7.5% of 40 repositories sample).

B.2 Impact of ShellCheck Reward Training

Based on findings from Appendix B.2, we constructed a checklist (Figure 5) capturing common Script Problems that we find possible to determine with simple localized checks, leaving mitigating further

Table 3: Identified environment setup failure patterns for zero-shot GPT-4o for 40 repositories (percentages are relative to full 40 repositories sample). **# Failure** means number of failed repositories which contain given pattern; **# Issues** — number of repositories with unresolved import issues. Note that each repository can contain multiple fault patterns.

Failure Pattern	Explanation	# Failure	# Issues
Script Problems			
Wrong Syntax	Syntax errors in the script.	4 (10%)	—
Dependencies Resolution Issue	Dependency manager can't resolve dependencies due to conflicting versions.	3 (7.5%)	—
Multiple Dep. Managers	Script uses both pip and Poetry.	2 (5%)	—
Wrong Python Binary	Script installs dependencies for a specific Python binary, but fails to configure the system to use that binary.	2 (5%)	1 (2.5%)
Missing System Package	Script doesn't install a system package required by repository dependencies.	1 (2.5%)	—
Non-existent Package	Script tries to install a package that does not exist on PyPI.	1 (2.5%)	—
Wrong Operation	Script executes a command that conflicts with the given base environment (e.g., tries to install Poetry even though it is already installed).	3 (7.5%)	—
Wrong Python Version	Script uses Python version conflicting with repository requirements.	1 (2.5%)	1 (2.5%)
Missing Dep. Group	Script does not install an optional dependency group required for development (e.g., `test`).	—	9 (22.5%)
No Editable Mode	Script installs the repository in non-editable mode not suitable for development (relevant for pip).	—	3 (7.5%)
Missing Configuration File	Script does not install dependencies from a configuration file in the repository (e.g., multiple `requirements-dev`, `requirements-docs`, etc.).	—	2 (5%)
Repository Problems			
Requirements Not Specified	Some packages used in the repository codebase are not specified in the configuration files.	—	10 (25%)
Poetry Lock Outdated	`poetry install` fails because the `poetry.lock` file must be regenerated first.	2 (5%)	—
Misconfigured PYTHONPATH	Local modules do not resolve correctly because the `PYTHONPATH` environment variable is not configured properly.	—-	2 (5%)
Eval Problems			
Dynamic Imports	Repository includes dynamic imports that cannot be resolved with static analysis.	—	5 (12.5%)
Eval Failure	Runtime failure of EnvBench evaluation suite, not associated with specific script or repository characteristics.	2 (5%)	—
Hardware Problems	Dependencies require hardware not available in the base environment (e.g., GPU).	1 (2.5%)	—

Checklist of Locally Verifiable Environment Setup Script Problems

Current script...

installs incorrect Python version?	[YES / NO / N/A]	Wrong Python Version
uses incorrect path to configuration file?	[YES / NO / N/A]	Wrong Syntax (conf)
misses optional development dependencies?	[YES / NO / N/A]	Missing Dep. Group
uses incorrect dependency manager?	[YES / NO]	Wrong Dep. Manager*
uses multiple dependency managers?	[YES / NO]	Multiple Dep. Managers
uses `pip install` command incorrectly?	[YES / NO / N/A]	Wrong Syntax (pip)
uses `poetry install` command incorrectly?	[YES / NO / N/A]	Wrong Syntax (Poetry)

Figure 5: Checklist used to detect locally verifiable Script Problems for environment setup. References failure patterns detailed in Table 3. Checklist questions are formulated such that YES indicates the presence of a failure pattern, NO indicates its absence, and N/A indicates that current pattern is not applicable to a given script (e.g., pip syntax is irrelevant if only Poetry is used). * While Wrong Dep. Manager did not appear in our exploration of GPT-4o scripts, we added it to the checklist as it was repeatedly observed for Qwen3-32B.

Script Problems, Repository Problems, and Eval Problems to future work. Next, we manually filled the checklist for scripts generated by Qwen3-32B in a zero-shot scaffold. Specifically, we considered both pretrained Qwen3-32B and Qwen3-32B-RL, a fine-tuned checkpoint from our preliminary experiments with ShellCheck reward (Section 3.2) that achieved close to perfect reward scores at the end of the training. Since we did not observe consistent improvements on EnvBench after training with ShellCheck reward (Section 4), with this study, we aimed to understand how exactly the training affects the properties of scripts for our end task, environment setup. We considered first 40 repositories in lexicographical order where the scripts from both checkpoints were available, same as in our study for GPT-4o (Appendix B.1). EnvBench evaluation results on this sample indicate a degradation after training: while the base Qwen3-32B model successfully set up a 5/40 repositories, after training, pass@1 drops to 1/40, with the number of failed repositories increasing from 13/40 to 23/40.

We present the results for base Qwen3-32B (pre-training) and for ShellCheck-reward-trained Qwen3-32B (post-training) in Figure 6. The most common problems faced by both base and ShellCheck-trained Qwen3-32B are Missing Dep. Group (30%-60%) and Wrong Syntax (pip) (5%-20%), mirroring our findings for GPT-4o (Appendix B.1). The ShellCheck-trained model shows deteriorated performance across most checklist questions compared to the base model. Combined with worse EnvBench evaluation results, this suggests that addressing ShellCheck-reported issues alone is insufficient to improve environment setup capabilities and can potentially even harm them. We use this analysis to design Heuristics reward (Appendix A.6).

C Train/Validation Performance

To rule out memorization from improvements of our RLVR-tuned models that were trained on a part of EnvBench, we present experimental results separately on the held-out validation set in Table 4 with pass@k and # Failed metrics converted to percentages to account for different sample sizes. From Table 4, we observe that LLM-as-a-Judge and Heuristics similarly retain substantial improvements over the base Qwen3-8B on the validation set, while ShellCheck shows no significant gains. Specifically, LLM-as-a-Judge displays strong performance close to the level of Qwen3-32B and GPT-4o, while Heuristics brings moderate gains, mirroring our findings on full EnvBench.

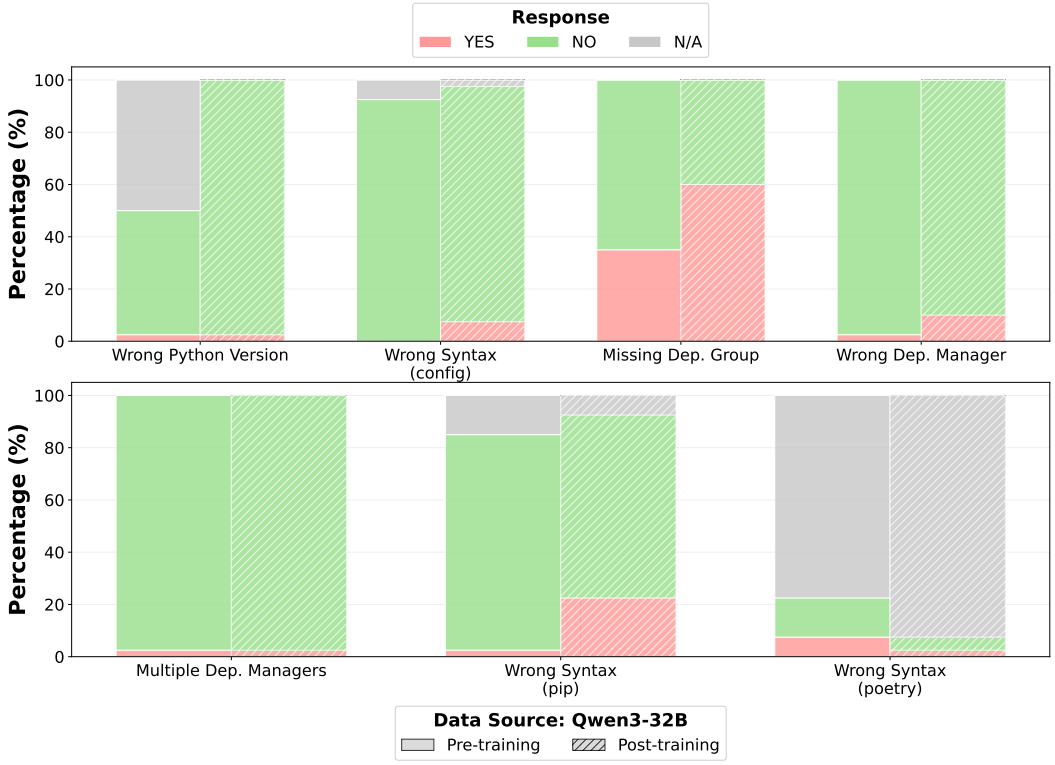

Figure 6: Results of applying the Script Problems checklist (Figure 5) to base Qwen3-32B (Pre-training) and a checkpoint trained with ShellCheck reward obtained in our initial experiments (Post-training).

Table 4: **Validation** split results for base models and our RLVR-tuned Qwen3-8B with various rewards. Total number of samples is 96. **pass@5** shows the number of successful samples (zero exit code and zero issues). **avg@5** shows mean ± std for the following metrics: **# Success** (average number of successful samples per run), **# Failed** (average number of samples where scripts finished with non-zero exit code), and **avgFixRate** (average ratio of resolved import issues per sample as compared to the evaluation run with empty setup script; for samples where scripts execute with non-zero exit codes, ratio is considered 0). The symbol ↑ indicates higher is better, while ↓ indicates lower is better. We provide percentages relative to the sample size in addition to all absolute numbers.

Model	Reward	pass@5	avg@5		
		# Success ↑	# Success ↑	# Failed ↓	avgFixRate ↑
Proprietary Baselines					
GPT-4o	—	7	4.4 ± 0.5	55.2 ± 1.5	$(23.96 \pm 13.2)\%$
GPT-4o-mini	—	2	1.8 ± 0.4	$\mathbf{47.6 \pm 3.2}$	$(22.62 \pm 1.31)\%$
Open-Weight Baselines					
Qwen3-32B	—	6	4.4 ± 1.3	56.0 ± 4.4	$(28.77 \pm 2.09)\%$
Qwen3-14B	—	3	1.0 ± 0.7	79.8 ± 4.1	$(10.06 \pm 4.38)\%$
Qwen3-8B	—	2	0.4 ± 0.9	85.4 ± 1.8	$(5.76 \pm 1.83)\%$
RLVR-tuned (ours)					
Qwen3-8B	LLM-as-a-Judge	**8**	$\mathbf{5.6 \pm 1.1}$	57.0 ± 2.9	$\mathbf{(30.79 \pm 2.13)\%}$
	Heuristics	3	1.0 ± 0.7	64.4 ± 3.6	$(17.15 \pm 1.43)\%$
	ShellCheck	1	1 ± 0	83.6 ± 2.3	$(6.88 \pm 1.62)\%$

D Additional Experiments

In this section, we describe a set of experiments that did not make it into the main text. These results highlight the practical challenges and training instabilities often encountered in RLVR, and are included for completeness and might be of interest to researchers working on similar setups.

D.1 Irreproducible High-Performing Shellcheck Checkpoint

During our experiments, one Shellcheck training run produced an exceptionally strong model checkpoint, achieving pass@5 = 18, average pass@1 = 9 ± 2, and avgFixRate = 25 ± 2. By all metrics, this checkpoint performed between the Heuristics and LLM-as-a-Judge models. We have not included this result in the main text, as we were unable to reproduce it despite multiple attempts. This run used a slightly different configuration, with fewer epochs and a shorter response length, but even after matching all settings, the result remained an outlier and could not be replicated. This highlights the inherent instability of RLVR training and underscores the importance of verifying results with multiple random seeds.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer:

   Justification: We provide detailed methodology (Section 3) and experimental results (Section 4) to support the claims made in the abstract and introduction.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer:

   Justification: We discuss the limitations of our study in Section 6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer:

Justification: The paper does not include theoretical results.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer:

   Justification: We provide a comprehensive experimental methodology in Section 3 and Appendix A.2 and open-source our replication package (`https://github.com/envsetup-rl-dl4c/envsetup-rl`) to facilitate reproduction.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer:

Justification: Our replication package is openly available: `https://github.com/envsetup-rl-dl4c/envsetup-rl`. We provide the training code, the model checkpoints, and full evaluation results.

Guidelines:

- The answer NA means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so "No" is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer:

   Justification: We provide a comprehensive experimental methodology in Section 3 and Appendix A.2.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer:

   Justification: We report the results with error bars over 5 evaluation runs.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer:

   Justification: We describe our resource setup in Section 3 and Appendix A.2.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer:

   Justification: We reviewed and followed NeurIPS Code of Ethics.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer:

    Justification: Our work targets improving the environment setup capabilities of general-purpose foundation models. We discuss potential positive impacts, such as assistance in the daily work of software developers and expansion of execution-based software benchmarks. Potential negative impacts include harms when the system is used as intended but is incorrect (e.g., hallucinated unsafe commands) and misuse to prepare environments for malware or privacy-invasive tooling. As partial mitigation, we recommend sandboxed execution and user

review before running model-generated commands. However, given the scope (advancing capabilities of open-source LLMs on the environment setup task), we consider broader societal impacts minimal.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer:

    Justification: We release trained Qwen3-8B checkpoints with enhanced environment setup capabilities and do not materially expand the model's general capabilities or misuse risk relative to the base model. We implement no additional safeguards beyond those introduced for the base model [Qwen Team, 2025] due to time and resource constraints, and do not release any new datasets or interactive demos. Details of the base model are in Qwen Team [2025].

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer:

Justification: We explicitly mention the creators of the assets used in the paper and follow the corresponding licenses.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer:

Justification: We release the training code, the Qwen3-8B checkpoints from our experiments, and their full evaluation results. We describe training and evaluation setup in Section 3 and Appendix A.2 and further provide detailed documentation in our replication package (`https://github.com/envsetup-rl-dl4c/envsetup-rl`).

Guidelines:

- The answer NA means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer:

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer:

Justification: The paper does not involve crowdsourcing nor research with human subjects.

Guidelines:

- The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does not impact the core methodology, scientific rigorousness, or originality of the research, declaration is not required.

Answer:

Justification: We propose a method to improve LLM capabilities for the environment setup task and use multiple LLMs in our experiments. In addition, we use an LLM in the proposed LLM-as-a-Judge reward. Both are described in Section 3.

Guidelines:

- The answer NA means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy (`https://neurips.cc/Conferences/2025/LLM`) for what should or should not be described.