PIPER: ON-DEVICE ENVIRONMENT SETUP VIA ONLINE REINFORCEMENT LEARNING

Anonymous authors
Paper under double-blind review

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 in automating this task. To address this limitation, we tune a specialized model for environment setup. We combine supervised fine-tuning for generating correct Bash scripts and Reinforcement Learning with Verifiable Rewards (RLVR) to adapt it to the task of environment setup. On EnvBench-Python, our method enables Qwen3-8B (a model runnable on consumer hardware) to perform on par with larger models—Qwen3-32B and GPT-4o. The training code and model checkpoints are available online: `https://github.com/PIPer-iclr/PIPer`.

1 INTRODUCTION

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 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., 2025b), 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 datasets (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., 2025) 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% (22 out of 329), achieved by GPT-4o in an agentic workflow.

We seek to improve small open-source models to democratize the usage of LLMs for environment setup. To this end, we analyze the environment setup scripts produced by strong LLMs on EnvBench and employ both supervised fine-tuning (SFT) and reinforcement learning (RL) to resolve found issues. The proposed method achieves more than $9\times$ improvement over the base model, being on par with the open-source model four times the size, and strong closed-source baselines. Specifically, our

contributions are: (1) the first application of online reinforcement learning with lightweight verifiable reward to environment setup, (2) on-device sized PIPER model performing on par with strong baselines offering a superior performance-cost ratio, and (3) a rigorous evaluation, demonstrating that the model trained with the proposed method generalizes across different datasets, indicating genuine scripting capability enhancement. To facilitate reproducibility and future research in this direction, we make our code, model weights, and generated scripts publicly available [1].

The rest of the manuscript is organized as follows. We describe the datasets used for training and evaluation in Section 2, motivate and describe the training approach in Section 3, describe how we set up the experiments in Section 4, and provide an overview of our experimental results in Section 5.

2 DATASET

The focus of our work is to democratize the use of LLMs for environment setup. To measure our progress in this task, we select two environment setup benchmarks, EnvBench (Eliseeva et al., 2025) and Repo2Run (Hu et al., 2025b). Also, to check how this training affects a broader set of tasks, we employ Terminal-Bench (The Terminal-Bench Team, 2025). In this section, we outline the specifics of each dataset we use—the inputs and outputs of the evaluated method, and the definition of task resolution.

EnvBench-Python comprises 329 Python repositories from GitHub. As an input, an 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. As an output, an environment setup approach should produce a shell script that installs all the needed dependencies in the base environment. The correctness of the environment setup script is evaluated by first executing it, and then invoking Pyright[2]—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.

Repo2Run comprises 420 Python repositories from GitHub with no overlaps with EnvBench-Python. The original work primarily focuses on an agentic setting, where an environment setup agent is granted access to the base environment with the repository through a terminal interface and other specialized tools. The agent is then expected to autonomously configure the repository by interacting with the environment. In contrast with static analysis-based metrics from EnvBench-Python, Repo2Run runs test collection via pytest[3] to verify the environment setup correctness. We include Repo2Run to verify that our experimental results transfer across different repositories and success criteria. We additionally adapt Repo2Run to settings beyond agentic, employing a more general task formulation similar to that of EnvBench-Python (discussed in detail in Section 4.2).

Terminal-Bench comprises 80 tasks focused on command-line environment configuration tasks (we use version 0.1.1 of the benchmark), evaluating AI agents' ability to handle real-world, end-to-end terminal operations, including compiling code, training models, and setting up servers. Each task consists of the problem described in natural language passed to an LLM, a Docker environment, and a test script to verify if the agent completed the task successfully. We use the original implementation[4] with multi-turn agentic scaffold Terminus 1. The success is determined by whether the agent can complete the specified terminal-based objective within a sandbox environment. We use Terminal-Bench to assess whether our training pipeline, designed primarily for single-turn Python package installation scenarios, generalizes to broader, out-of-distribution, multi-turn terminal command execution tasks beyond dependency management.

3 METHOD

To train the model, we employ a two-stage process widely adopted in the literature (Liu et al., 2025b; Yoshihara et al., 2025; Golubev et al., 2025). First, we tune the model in a supervised manner on the

[1] Replication Package: `https://github.com/PIPer-iclr/PIPer`

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

[3] `https://pytest.org`

[4] `https://github.com/laude-institute/terminal-bench`

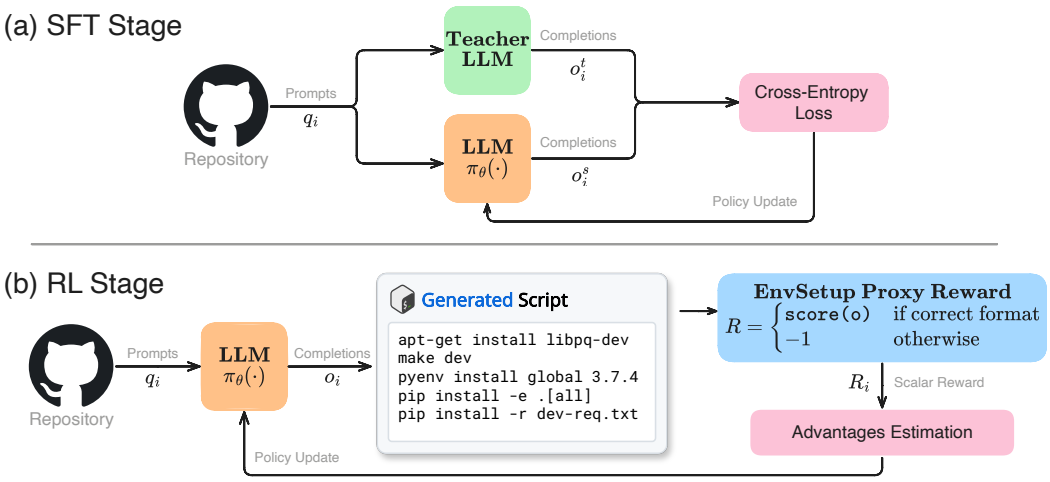

Figure 1: Overview of the proposed training pipeline. (a) **SFT training**: For the i-th sample (a repository), both teacher and student LLMs receive the prompt q_i, which includes the task description and repository context. They generate completions o_i^t and o_i^s, respectively, expected to contain a shell script. The student model's weights are updated by minimizing the cross-entropy loss between its output distribution and the teacher's completion. (b) **RL training**: For each sample, LLM π_θ generates a completion o_i, expected to contain a shell script. The completion is evaluated by a rule-based reward function R, which outputs a score R_i. The REINFORCE++ algorithm then updates the LLM weights using the rewards R_i and responses o_i.

executable scripts sampled from the larger model of the same family. Then, we run one more stage of RL training, to refine the capabilities of the model after the SFT update. We employ the RLVR technique since it has been reported to show promising results on tasks from the SE domain (Luo et al., 2025; Golubev et al., 2025). In our notation throughout this paper, we use q to denote prompts provided to the model, o to represent model responses, s to refer to shell scripts extracted from model outputs, and π_θ to denote the model with parameters θ. We use regular expressions to extract the shell script from the model outputs, and if parsing fails, we consider s to be empty. The schematic representation of the training stages is illustrated in Figure 1. Further, we introduce the details of the method. In Section 3.1 we discuss the SFT training, and in Section 3.2 we introduce the RL training.

3.1 Supervised Fine-Tuning

The supervised fine-tuning involves training a model on a set of data points that are considered to be ground truth. However, for the task of environment setup, and for the hard repositories specifically, it is a costly task to obtain such ground truth scripts. The authors of EnvBench provide only a small number of scripts generated by experts, and even strong models are solving only a small portion of the dataset (Eliseeva et al., 2025). Due to this, we employ distillation (Hinton et al., 2015), a technique where the small model (called Student) learns to imitate the behavior of a larger model (called Teacher). Our setup is shown in Figure 1(a) and detailed below.

We implement the SFT stage using executable scripts collected during the evaluation of a larger Qwen3-32B model. We first collect samples $\{q_i, o_i^t\}$ from evaluation rollouts. Then we filter out the samples where o_i^t doesn't contain a script, or the script results in a non-zero exit code. Finally, we select 2,500 pairs $\{q_i, o_i^t\}$ at random to form the distillation dataset. The student model π_θ is trained on this dataset in a supervised manner without further changes or masking. Since these samples originate from a different, larger model rather than π_θ, there is a potential distributional shift between the generated solutions and our model's natural output distribution, which can affect the generalization capabilities of the model (Shenfeld et al., 2025; Chu et al.). However, this approach allows us to leverage higher-quality executable solutions that demonstrate successful task completion patterns. The resulting SFT checkpoint serves as the foundation for the subsequent RLVR training.

3.2 Reinforcement Learning

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 turn to lightweight execution-free LLM-as-a-Judge reward (denoted R_{LLM}), which serves as a verifiable reward by mimicking rule-based evaluation criteria. The general scheme is presented in Figure 1(b).

To design the reward, 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 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%). A detailed description of the analysis process and all findings are presented in Appendix B.

The reward R_{LLM} takes in the extracted script s along with a comprehensive context for the corresponding repository and emulates the EnvBench evaluation suite. The instruction for the judge is motivated by our findings of typical errors, and prompts it to predict the exit code from the shell script execution and the number of Pyright issues (`num_issues`). Further implementation details could be found in Appendix A.4. Formally, 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 Experiments Setup

4.1 Training setup

Data. 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. However, to further ensure the absence of memorization, we also (1) reserve 96 repositories as a held-out validation set, using only the remaining 228 repositories for training and (2) evaluate performance on external benchmarks beyond EnvBench. We compare the results on the train and validation sets in Appendix C and find no strong indication of memorization. Due to technical issues, we omit five repositories from EnvBench from our training and validation sets.

Models. We select Qwen3-8B as our base model for its strong performance on SE tasks and reasonable compute requirements (Qwen Team, 2025). Qwen models also show consistent improvements with RLVR training compared to other model families (Gandhi et al., 2025). We leave the exploration of other model families and model sizes to future work. We use non-thinking mode because reasoning traces are often long and increase the GPU memory requirements as well as the training duration (Sui et al., 2025).

Scaffold. Our experiments follow the 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.

SFT Framework and Hyperparameters. We employ the LLamaFactory framework (Zheng et al., 2024) using a full-weight training approach. We perform the training with cross-entropy loss on a single H200 GPU for five epochs, without early stopping. We use the AdamW optimizer (Loshchilov and Hutter) and the effective batch size of 16. We reserved 5% of the samples for validation and did not observe signals of overfitting. Comprehensive hyperparameter setup and training details are listed in Appendix A.2.1.

RL Frameworks and Hyperparameters. We use the VeRL framework (Sheng et al., 2024). All our RL training runs are executed on 4xH200 GPUs with all weights optimized by the REINFORCE++ (Hu et al., 2025a) algorithm. A more exhaustive comparison with GRPO and GRPO-like objectives is left for future work. We set the batch size of 64 and the number of epochs to 15, yielding 45 training steps. We truncate the prompts longer than 30,000 tokens and allow the model to generate up to 4,096 tokens in response. We use vLLM (Kwon et al., 2023) as the rollout engine and set sampling parameters to the values recommended in the 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. We use GPT-4.1 as the backbone LLM for the judge. Comprehensive hyperparameter setup and training details are listed in Appendix A.2.2.

4.2 Evaluation Setup

EnvBench-Python (Eliseeva et al., 2025). For EnvBench, we extend the original work with three additional metrics. The first one is *pass@5*—the binary measure of success across 5 attempts for each datapoint. It is equal to 1 for a given repository, if at least once in 5 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. To calculate this, we first take the percentage of issues fixed for each repository and then average this number 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 also report *# Failed*—number of repositories where the scripts resulted in a non-zero exit code. All metrics apart from pass@5 are reported averaged over five runs. We use the same base environment, zero-shot scaffold and prompt as during training (Section 4.1), with evaluation infrastructure available in our replication package.

Repo2Run (Hu et al., 2025b). For Repo2Run, we also use the *pass@5* metric. Success is determined by running test collection via pytest: if there are no collection errors, the setup is considered successful. We do not use the agentic setting from original work and instead employ the same base environment as the EnvBench.

Terminal-Bench (The Terminal-Bench Team, 2025). For Terminal-Bench, which is a more challenging benchmark, especially for smaller models, we use the *pass@10* metric. The benchmark provides custom evaluation commands for each data point.

Baselines. We compare the trained models against multiple general-purpose LLMs. We evaluate three closed-source OpenAI models: GPT-5, GPT-4o, and GPT-4o-mini. We also assess multiple models from the Qwen3 family (8B, 14B, and 32B parameters) to understand how our approach compares across different model scales. All Qwen3 models are evaluated in non-thinking mode for consistency.

5 Results

5.1 Training Dynamics

Training dynamics of base and SFT model with the LLM-based reward described in Section 3.2 are depicted in Figure 2. The reward function returns values from the $[-1, 1]$ range, where -1 indicates malformatted scripts, and 1 indicates perfect performance.

Table 1: Results on Repo2Run and Terminal-Bench for base models and our tuned Qwen3-8B. For Repo2Run, success is determined as a zero exit code and no test collection errors. For Terminal-Bench, success is determined by per-sample evaluation commands. Our PIPER model achieves the best performance on Repo2Run. However, SFT-based models underperform on Terminal-Bench's multi-turn setting.

Model	Repo2Run	Terminal-Bench
	pass@5	pass@10
GPT-5	❶ 106	❶ 45
GPT-4o	67	❷ 25
GPT-4o-mini	84	19
Qwen3-32B	71	❸ 23
Qwen3-14B	64	14
Qwen3-8B	32	8
PIPER	❷ 103	4
PIPER$^{\text{RL-only}}$	77	9
PIPER$^{\text{SFT-only}}$	❸ 98	2

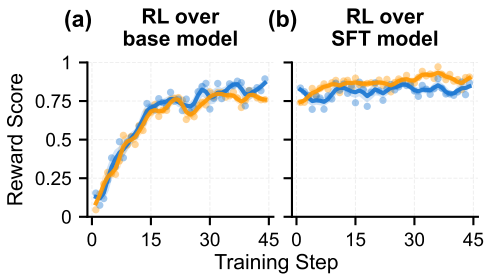

Figure 2: RLVR training dynamics with the proxy rewards described in Section 3.2. Raw datapoints are shown as semi-transparent dots, with Gaussian-smoothed curves overlaid to highlight trends. **Blue** shows average reward on the training set; **orange** shows average reward on the validation set. The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

The base model exhibits formatting compliance but fails to satisfy the meaningful criteria imposed by the reward. This is evident from almost zero **validation** scores at step 0, which are close to the minimum values achievable with the correct formatting. On the other hand, the SFT checkpoint from the start produces high-quality scripts that are correctly formatted and highly assessed by the judge.

We observe a steady initial increase for both **training** and **validation** sets, which then slows for the base model, and plateaus for the SFT model. The substantial differences in validation reward scores between step 0 and step 45 suggest that RLVR training successfully steers the models to better adhere to the criteria imposed by the reward. In addition, we do not observe strong overfitting: there is only a small gap between **training** and **validation** reward scores.

5.2 Evaluation Results

The evaluation results on EnvBench are presented in Table 2. While the GPT-5 frontier model claims the first place, the proposed PIPER model is competitive with both strong open-source (Qwen3-32B) and closed-source (GPT-4o) baselines. With respect to all reported metrics on EnvBench, it comes with a small gap or is on par with these strong competitors. The outlier metric here is the surprisingly low *# Failed* of the GPT-4o-mini model, which claims the second place in the rating, and is significantly better than GPT-4o. We leave a thorough investigation of this phenomenon to future work.

To assess the cost-performance ratio of PIPER, we compare the baselines with respect to both performance and inference cost. We take prices per 1M generated tokens as the cost of a model. For OpenAI models, we take official API prices[5], and for the Qwen3 model family, we take costs from the Alibaba Cloud website[6]. The costs are reported as of 22.09.2025. Figure 3(b) indicates comparable-or-better performance relative to baselines at a fraction of their cost; PIPER can also run on local machines, further reducing cost.

The ablation of RL and SFT phases, also shown in Table 2, shows the necessity of the two-stage pipeline. While both SFT and RL checkpoints outperform the base model, pushing its *avg@5* performance from 2.6 to 13 (SFT) or 11.8 (RL), they both are significantly worse than the checkpoint yielded by the combined training.

[5] https://platform.openai.com/docs/pricing

[6] https://www.alibabacloud.com/help/en/model-studio/models

Table 2: EnvBench evaluation results for base models and PIPER with various training setups. The 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	pass@5	avg@5		
	# Success ↑	# Success ↑	# Failed ↓	avgFixRate ↑
GPT-5	❶ 43	❶ 25 ± 3	❶ 131 ± 5	❶ $(32.7 \pm 3.3)\%$
GPT-4o	❷ 29	❷ 19 ± 2	194 ± 6	❷ $(28.0 \pm 1.0)\%$
GPT-4o-mini	15	9.6 ± 1.3	❷ 166 ± 6	$(22.6 \pm 1.5)\%$
Qwen3-32B	❷ 29	❸ 16.2 ± 1.3	207 ± 6	$(25.1 \pm 1.3)\%$
Qwen3-14B	17	5.6 ± 1.1	268 ± 10	$(9.95 \pm 0.81)\%$
Qwen3-8B	8	2.6 ± 1.5	294 ± 2	$(4.4 \pm 1.2)\%$
PIPER	❸ 27	❷ 19 ± 3	❸ 183 ± 3	❸ $(27.2 \pm 1.2)\%$
PIPER$^{\text{SFT-only}}$	25	13.0 ± 1.0	192 ± 7	$(23.6 \pm 1.4)\%$
PIPER$^{\text{RL-only}}$	19	11.8 ± 0.8	205 ± 5	$(25.2 \pm 1.2)\%$

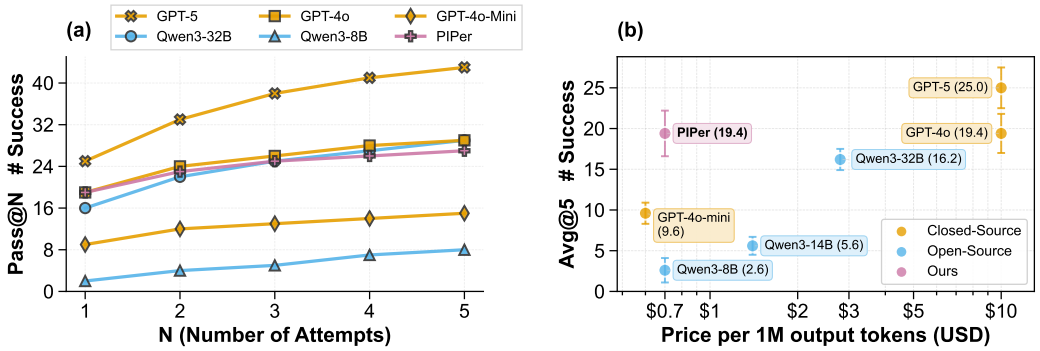

Figure 3: Performance analysis of environment setup models on EnvBench-Python. **(a)** Pass@N performance showing how model success rates improve with multiple attempts ($N = 1$ to 5). Our PIPER model (shown with cross markers) achieves performance comparable to much larger models like GPT-4o and Qwen3-32B, while substantially outperforming the base Qwen3-8B model. **(b)** Cost-performance tradeoff analysis comparing average *pass@1* performance (averaged over five runs) against price per 1M output tokens (USD).

Finally, in the Figure 3(a), we explore how the model results improve with multiple attempts. While the scaling of PIPER is slower than that of the strong baselines, it is still able to beat them at cost parity. For example, *pass@3* of the proposed model is higher than the *pass@2* of both GPT-4o and Qwen3-32B models. Also, *pass@5* of PIPER (27) is higher than *pass@1* of GPT-5 (25), while the cost of inference is more than 14 times lower.

5.3 GENERALIZATION

To ensure that the results of the evaluation are not based on overfitting, we separately evaluate our model on the evaluation subset of EnvBench and on two additional datasets. We detail the scores on

the evaluation subset of EnvBench in Appendix C, but notice that the model yields the results better or on par with the strong baselines. This confirms that there are no strong signs of memorization.

Table 1 shows the evaluation of the PIPER on Repo2Run and Terminal-Bench. On Repo2Run, which shares similar single-turn Python environment setup characteristics with EnvBench, all our trained models substantially outperform the base Qwen3-8B (32 success cases), with PIPER achieving the best results (103 success cases) and even surpassing larger models like Qwen3-32B (71 success cases) and GPT-4o-mini. However, on Terminal-Bench, which requires multi-turn agentic interactions for system configuration tasks, we observe a different pattern: while the PIPER$^{\text{RL-only}}$ model shows modest improvement (8 to 9 success cases), PIPER$^{\text{SFT-only}}$ (2) actually underperforms the base model, with united training procedure of PIPER showing a slight recovery (4). This suggests that while SFT improves single-turn performance, it struggles with the multi-turn interactions required by Terminal-Bench. These cross-benchmark results demonstrate that our proxy reward-based RLVR training develops transferable shell scripting capabilities, with the RL component providing more robust generalization across diverse interaction paradigms than supervised fine-tuning alone.

6 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 for training because its construction process explicitly prioritizes challenging repositories, providing a diverse learning signal.

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., 2025; Zhang et al., 2025; Guo et al., 2025). Existing works use general-purpose LLMs as backbones, and many workflows include execution of intermediate agent outputs (Eliseeva et al., 2025; Milliken et al., 2025; Bouzenia and Pradel, 2025; Hu et al., 2025b; Vergopoulos et al., 2025; 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., 2025; 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., 2025a), 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). Recent comparative studies of RL and SFT training approaches and their combinations have revealed both synergistic improvements (Liu et al., 2025b; Yoshihara et al., 2025) and potential degradation of final model performance (Chen et al., 2025a), while also demonstrating that SFT alignment can impair models' generalization capabilities (Shenfeld et al., 2025; Liu et al., 2025b; Wu et al., 2025).

7 Limitations and 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 the lightweight LLM-based reward function that allows 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.

8 Conclusion

We presented PIPER—a strong on-device-sized model for environment setup. It is trained with a two-stage pipeline without ground truth data, with SFT distillation to teach the model to write correct scripts, and RLVR to further improve the environment setup capabilities. We use a lightweight reward that mimics a ground truth execution check with the LLM-as-a-Judge technique to lift strong infrastructure requirements for the direct environment feedback. The resulting model performs on par or better than several times more expensive models, such as GPT-4o and Qwen3-32B. Importantly, our findings extend beyond environment setup. The trained models maintain reasonable performance on the out-of-distribution Terminal-Bench in an agentic scaffold, indicating genuine improvement of terminal manipulation capabilities rather than task-specific overfitting. Our replication package with training code and trained model checkpoints is available online: `https://github.com/PIPer-iclr/PIPer`.

9 Reproducibility Statement

To ensure full reproducibility of our results, all model checkpoints mentioned in this paper are publicly available, including PIPER$^{\text{RL-only}}$, PIPER$^{\text{SFT-only}}$, and the final PIPER model, along with all raw evaluation results from our experiments (`https://huggingface.co/PIPer-iclr`). The complete codebase used for SFT training, RL training, and evaluation is available in our dedicated repository (`https://github.com/PIPer-iclr/PIPer`), which contains all configuration files and implementation details. Upon acceptance of this work, we will also publish the complete training run logs to provide full transparency into the training process and enable detailed analysis of our experimental procedures.

10 Acknowledgements

We acknowledge the use of LLMs for text polishing and minor language improvements throughout this manuscript. All technical content, ideas, and substantial writing remain the original work of the authors.

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.

Hardy Chen, Haoqin Tu, Fali Wang, Hui Liu, Xianfeng Tang, Xinya Du, Yuyin Zhou, and Cihang Xie. Sft or rl? an early investigation into training r1-like reasoning large vision-language models. *arXiv preprint arXiv:2504.11468*, 2025a.

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 2025b. 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.

Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. Sft memorizes, rl generalizes: A comparative study of foundation model post-training. In *Forty-second International Conference on Machine Learning*.

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.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

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.

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*.

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. Tulu 3: Pushing frontiers in open language model post-training, 2025. URL https://arxiv.org/abs/2411.15124.

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, 2025a. URL https://arxiv.org/abs/2503.20783.

Zihan Liu, Zhuolin Yang, Yang Chen, Chankyu Lee, Mohammad Shoeybi, Bryan Catanzaro, and Wei Ping. Acereason-nemotron 1.1: Advancing math and code reasoning through sft and rl synergy. *arXiv preprint arXiv:2506.13284*, 2025b.

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.

Idan Shenfeld, Jyothish Pari, and Pulkit Agrawal. Rl's razor: Why online reinforcement learning forgets less. *arXiv preprint arXiv:2509.04259*, 2025.

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.

The Terminal-Bench Team. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025. URL https://github.com/laude-institute/terminal-bench.

Konstantinos Vergopoulos, Mark Niklas Müller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. *arXiv preprint arXiv:2503.07701*, 2025.

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.

Yongliang Wu, Yizhou Zhou, Zhou Ziheng, Yingzhe Peng, Xinyu Ye, Xinting Hu, Wenbo Zhu, Lu Qi, Ming-Hsuan Yang, and Xu Yang. On the generalization of sft: A reinforcement learning perspective with reward rectification. *arXiv preprint arXiv:2508.05629*, 2025.

Hiroshi Yoshihara, Taiki Yamaguchi, and Yuichi Inoue. A practical two-stage recipe for mathematical llms: Maximizing accuracy with sft and efficiency with reinforcement learning. *arXiv preprint arXiv:2507.08267*, 2025.

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.

Yaowei Zheng, Richong Zhang, Junhao Zhang, YeYanhan YeYanhan, and Zheyan Luo. Llamafactory: Unified efficient fine-tuning of 100+ language models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pages 400–410, 2024.

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
```

A.2 Training Details

A.2.1 SFT Training

We show the hyperparameters used in our SFT training in Table 3. We fine-tune the Qwen3-8B model for 5 epochs with a learning rate of 5×10^{-5} using the AdamW optimizer with cosine scheduling, weight decay of 0.01, gradient accumulation of 4 steps, and batch size of 4. Scripts, selected for the training, cover $227/228$ unique repositories from the train split with a median sample size of 11 for each repository. Training is performed with bfloat16 precision, with a 5% validation split for evaluation.

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 context variables are the same as theirs. Repository context is collected by executing a fixed set of commands within the repository in the target Docker environment.

Table 3: SFT parameters for LLaMA-Factory

Parameter	Value
Training Settings	
Epochs	5
Learning Rate	5e-5
Weight Decay	0.01
Optimizer	AdamW
LR Scheduler	Cosine
Gradient Accumulation Steps	4
Warmup Ratio	0.1
Max Grad Norm	1.0
Batch Size	4
Precision & Optimization	
Dtype	bfloat16
FlashAttention-2	enabled
Evaluation & Logging	
Validation Split	0.05
Early Stopping	—

Table 4: RL parameters for VeRL

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
Learning Rate	5e-6
Gradient Clipping	1.0
Total Steps	45
RL Settings	
Algorithm	Reinforce++
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.2.2 RL Training

We show the hyperparameters used in our RL training in Table 4. 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.

A.3 Evaluation Details

EnvBench. 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.

Repo2Run. As Repo2Run replication package only includes code for inference of the proposed Repo2Run agent, we extend EnvBench evaluation suite to support repositories and success check (test collection via pytest) from Repo2Run. We use the same zero-shot scaffold and prompts as for EnvBench, detailed in Appendix A.1. The resulting evaluation suite is available in our replication package.

Terminal-Bench. We use the original implementation to run the evaluation on Terminal-Bench. We use Terminus 1 scaffold and version 0.1.1 of the benchmark.

A.4 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).

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,

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

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 zero-shot context, and (2) replacing the LLM-as-a-Judge with an LLM Agent equipped with tools for repository exploration. Neither approach led to a noticeable improvement in model performance. Consequently, we adopted the simplest and most robust configuration for our main experiments.

B Empirical Study of Environment Setup 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 5, 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).

C Train/Validation Performance

To rule out memorization from improvements of our models that were trained on a part of EnvBench, we present experimental results separately on the held-out validation set in Table 6. From Table 6, we observe that all PIPER variants retain substantial improvements over the base Qwen3-8B on the validation set. Similarly, the performance of PIPER on the validation set is comparable to Qwen3-32B and GPT-4o, with either second-best or third-best results across all the considered metrics. GPT-5 remains the strongest among the considered baselines. Mirroring our findings on full EnvBench, SFT-only and RL-only checkpoints show lower per-run metrics than PIPER that combines both stages.

Table 5: 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 (45.5%)
Poetry Lock Outdated	`poetry install` fails because the `poetry.lock` file must be regenerated first.	2 (12.5%)	—
Misconfigured PYTHONPATH	Local modules do not resolve correctly because the `PYTHONPATH` environment variable is not configured properly.	—-	2 (9.09%)
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 (12.5%)	—
Hardware Problems	Dependencies require hardware not available in the base environment (e.g., GPU).	1 (6.25%)	—

Table 6: **Validation** split results for base models and PIPER model variations. 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.

Model	pass@5	avg@5		
	# Success ↑	# Success ↑	# Failed ↓	avgFixRate ↑
GPT-5	❶ 10	❶ 7.4 ± 1.5	❶ 36.0 ± 3.1	❶ $(41.4 \pm 2.8)\%$
GPT-4o	❸ 6	❸ 4.8 ± 1.1	60.4 ± 1.5	❸ $(26.5 \pm 1.1)\%$
GPT-4o-mini	3	2.0 ± 0.7	❷ 50.8 ± 2.0	$(20.4 \pm 0.8)\%$
Qwen3-32B	❷ 7	❸ 4.8 ± 0.4	62.6 ± 2.3	$(25.3 \pm 1.4)\%$
Qwen3-14B	4	1.8 ± 0.4	81.6 ± 1.7	$(10.4 \pm 1.5)\%$
Qwen3-8B	1	0.2 ± 0.4	89.6 ± 1.1	$(3.2 \pm 1.9)\%$
PIPER	❸ 6	❷ 5.2 ± 0.8	❸ 54.2 ± 1.6	❷ $(30.0 \pm 1.9)\%$
PIPER$^{\text{SFT-only}}$	❸ 6	3.2 ± 0.4	57.2 ± 1.3	$(24.6 \pm 2.3)\%$
PIPER$^{\text{RL-only}}$	❸ 6	3.8 ± 0.8	63.0 ± 3.1	$(24.6 \pm 1.9)\%$