Training Software Engineering Agents and Verifiers with SWE-Gym

Jiayi Pan [*1] **Xingyao Wang** [*2] **Graham Neubig** [3] **Navdeep Jaitly** [4] **Heng Ji** [2] **Alane Suhr** [†1] **Yizhe Zhang** [†4]

Abstract

Progress in agents for software engineering has been limited by the lack of training environments that both include rigorous verification for reinforcement learning and cover the expansive tasks encountered in real-world repository-level engineering. We present SWE-Gym, the first environment for training real-world software engineering agents, and use it to train language model (LM) agents that achieve state-of-the-art open results on the popular SWE-Bench dataset. SWE-Gym contains 2,438 real-world Python tasks, and when we fine-tune a 32B LM (Qwen-2.5-Coder) on agent-environment interaction trajectories sampled from it, two LM-powered agent scaffolds (OpenHands, Wang et al. 2024c and MoatlessTools, Örwall 2024) achieve gains of up-to 19% absolute accuracy on the popular SWE-Bench Verified and Lite test sets. Moreover, SWE-Gym enables inference-time scaling through verifiers trained on agent trajectories. These verifiers identify the most promising solutions via best-of-n selection; together with our learned agents, they reach 32.0% and 26.0% on SWE-Bench Verified and Lite, respectively. To facilitate further research, we release SWE-Gym, fine-tuned models, and agent trajectories [1].

1. Introduction

Language models (LMs) have demonstrated remarkable capabilities in automating software engineering tasks, as shown in SWE-Bench (Jimenez et al., 2024), where models are tasked to develop test-passing solutions for real-world GitHub issues using provided codebases and executable environments. While LM agent approaches have shown significant performance gains through improved agent-computer interfaces and prompting strategies (Yang et al., 2024; Wang et al., 2024c), open advances have been limited by their reliance on proprietary models, rather than improving the underlying LM.

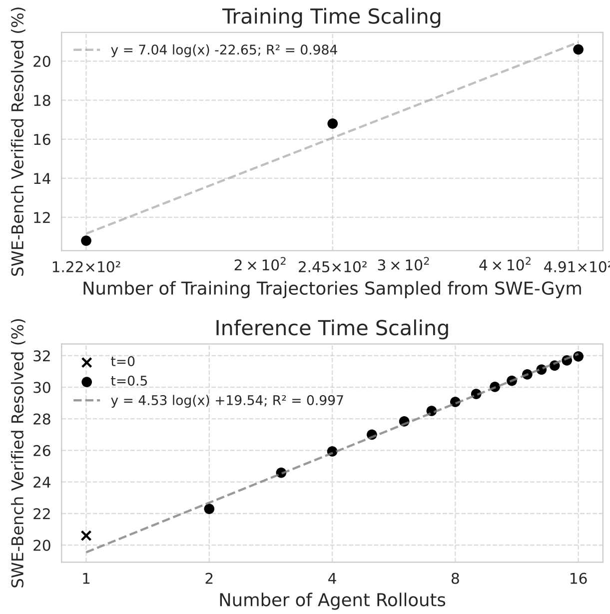

Figure 1: SWE-Gym enables scalable improvements for software engineering agents. **Top**: Training time scaling shows consistent performance improvements as we obtain more training trajectories, with no signs of saturation at 491 trajectories. This indicates that agent performance is currently bounded by compute for trajectory sampling rather than SWE-Gym's size. Evaluated at temperature $t = 0$. **Bottom**: Inference time scaling demonstrates roughly logarithm gains against number of sampled solutions, where we generate multiple candidates per task and select the best one using a verifier trained on SWE-Gym. $t = 0$ is used as the first hypothesis to be consistent with top figure and is excluded from regression, while later rollouts use $t = 0.5$.

Unlike other domains where supervised fine-tuning and reinforcement learning have significantly improved LM capabilities – as shown in general chat (Ouyang et al., 2022), mathematical reasoning (Shao et al., 2024; Yuan et al., 2024), and web navigation (Pan et al., 2024) – software

[*] Equal contribution. [†] Equal supervision [1]UC Berkeley [2]UIUC [3]CMU [4]Apple. Correspondence to: Jiayi Pan <jiayipan@berkeley.edu>, Xingyao Wang <xingyao6@illinois.edu>, Alane Suhr <suhr@berkeley.edu>, Yizhe Zhang <yizzhang@apple.com>.

[1] https://github.com/SWE-Gym/SWE-Gym

Dataset (split)	Repository-Level	Executable Environment	Real task	# Instances (total)	# Instances (train)
CodeFeedback (Zheng et al., 2024b)	✗	✗	✓	66,383	66,383
APPS (Hendrycks et al., 2021a)	✗	✓	✓	10,000	5,000
HumanEval (Chen et al., 2021)	✗	✓	✓	164	0
MBPP (Tao et al., 2024)	✗	✓	✓	974	374
R2E (Jain et al., 2024)	✓	✓	✗	246	0
SWE-Bench (train) (Jimenez et al., 2024)	✓	✗	✓	19,008	19,008
SWE-Gym Raw	✓	✗	✓	66,894	66,894
SWE-Bench (test) (Jimenez et al., 2024)	✓	✓	✓	2,294	0
SWE-Gym	✓	✓	✓	2,438	2,438

Table 1: SWE-Gym is the first publicly-available training environment combining real-world software engineering tasks from GitHub issues with pre-installed dependencies and executable test verification for training software engineering agents. **Repository-level**: whether solving each task instance requires repository context; **Executable Environment**: whether each instance in the dataset comes with an executable environment with all relevant dependencies pre-installed; **Real task**: whether the instruction for each instance is collected from human developers.

engineering agents lack suitable training environments. Creating such environments is particularly challenging as real-world software tasks require maintaining complex repository states with specific dependency versions and ensuring reproducible test environments across platforms, among other challenges. These challenges are reflected in the existing resources (Tab. 1). For example, the SWE-Bench (Jimenez et al., 2024) training split lacks executable environments and reward signals, R2E (Jain et al., 2024) uses synthetic instructions that are very far from real-world problems, while datasets such as CodeContests (Li et al., 2022) and APPS (Hendrycks et al., 2021a) focus only on isolated tasks rather than realistic repository-level coding problems.

To bridge this gap, we present SWE-Gym, the **first training environment** combining real-world software engineering tasks from GitHub issues with pre-installed dependencies and executable test verification[2]. SWE-Gym contains 2,438 Python tasks sourced from 11 popular open-source repositories (Tab. 2), providing useful environments for training LMs as agents and verifiers.

We demonstrate that **SWE-Gym allows training of LMs as agents**. Based on OpenHands (Wang et al., 2024c), an agent scaffold for general-purpose software development (§5.1), we show that we can fine-tune a 32B Qwen-2.5 coder model (Hui et al., 2024b) using less than 500 agent-environment interaction trajectories sampled from SWE-Gym, and achieve substantial absolute improvements of $+\mathbf{12.3}\%$ (to 15.3%) and $+\mathbf{13.6}\%$ (to 20.6%) on SWE-Bench Lite and SWE-Bench Verified respectively (§3.2).

SWE-Gym is effective across agent scaffolds. In another agent scaffold based on a SWE-Bench specialized workflow (Moatless, Örwall 2024, §5.1), we observe an improvement to **19.7%** (32B model) and **10.0%** (7B model) on SWE-Bench Lite through self-improvement, where the initial open-weight LM interacts with SWE-Gym, receives reward from it, and learns to improve the policy through rejection sampling fine-tuning.

SWE-Gym allows for training verifier models that enable inference-time scaling. By leveraging both successful and failed agent trajectories – determined through test suite executions – we train a verifier model (i.e., outcome-supervised reward model) that predicts each agent trajectory's probability of success. The trained verifier enables inference-time scaling by sampling multiple agent trajectories and selecting the best one based on predicted rewards. This further improves the resolve rate to 32.0% (+11.4% absolute) on SWE-Bench Verified (§4.1.1, Fig. 1 bottom) and 26.0% on SWE-Bench Lite (§4.1.2), establishing a new state-of-the-art among systems with publicly accessible weights (Tab. 9).

Lastly, as shown in Fig. 1, **our baseline training and inference-time scaling methods on SWE-Gym yield continuously improved results with increasing compute.** In the training phase, performance scales with the number of sampled trajectories up to our current limit of 491 trajectories, suggesting that performance is currently limited by sampling compute rather than SWE-Gym's size. Similarly, using the agent and verifier trained by SWE-Gym, the bottom panel shows that adding more compute during the inference time steadily improves the results.

We release SWE-Gym, fine-tuned models, and agent trajectories to facilitate further research in this direction: `https://github.com/SWE-Gym/SWE-Gym`.

2. SWE-Gym Environment

SWE-Gym comprises 2,438 real-world software engineering tasks sourced from pull requests in 11 popular Python repositories, with pre-configured executable environments and validated test cases, constructed in close alignment with

[2]We discuss a concurrent work by Golubev et al. (2024) in §A.

Category	Metric (Mean)	SWE-Gym	SWE-Gym Lite
Size	# Instances	2,438 (2,294)	230 (300)
	# Repos	11 (12)	11 (12)
Issue Text	Length by Words	239.8 (195.1)	186.16 (175.9)
Codebase	# Non-test Files	971.2 (2944.2)	818.8 (2988.5)
	# Non-test Lines	340675.0 (363728.4)	340626.2 (377562.4)
Gold Patch	# Lines edited	69.8 (32.8)	10.6 (10.1)
	# Files edited	2.5 (1.7)	1.0 (1.0)
	# Func. edited	4.1 (3.0)	1.4 (1.34)
Tests	# Fail to Pass	10.0 (9.0)	2.04 (3.5)
	# Total	760.8 (132.5)	99.9 (85.2)

Table 2: Statistics about our SWE-Gym compared to SWE-Bench test split (in parenthesis).

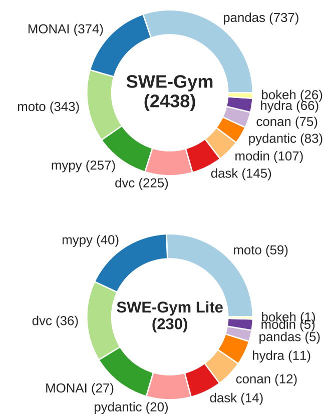

Figure 2: Repository distribution of SWE-Gym instances.

SWE-Bench (Jimenez et al., 2024). We introduce two variants of the dataset. The standard SWE-Gym includes full 2,438 tasks, while SWE-Gym Lite is a subset of 230 tasks.

To support future research in automatic dataset synthesis, we additionally release all Python GitHub issues we crawled that were either unprocessed or failed the validation process, which we refer to as SWE-Gym Raw. SWE-Gym Raw includes 66,894 instances spanning 357 Python repositories.

2.1. Collect Training Instances

Repository Collection. We used SEART Github search [3] to filter a list of initial repositories. We selected Python repos that were created before 7/1/2022, has more than 500 stars, with at least 300 lines of code, more than 500 pull requests (PRs), and 100 contributors.

Extract Training Instances from Repositories. We use SWE-Bench's instance extraction script to convert these repositories into SWE-Bench-style instances, each corresponding to a Github issue with a set of unit tests. Eventually, we collected 358 repositories, totaling 64,689 instances. We refer to this dataset as SWE-Gym Raw, which is over three times larger than the 19K instances gathered in previous work (Jimenez et al., 2024) and includes nearly ten times as many repositories.

Version Training Instances. Out of the 357 total repositories collected, we focus on a subset of 11 repositories with high number of instances for annotation. We generalize SWE-Bench's versioning script to support versioning via script execution, and semi-automatically collect versions for each instance based on information available in the repository (e.g., `pyproject.toml`, git tag, etc).

[3] https://seart-ghs.si.usi.ch/

Setup Executable Environments for Each Training Instances. An environment with pre-installed dependencies is essential for developing software engineering agents, as it enables unit test feedback and allows agents to focus on task solving instead of dependency management. However, configuring dependencies for a specific version of the codebase can be particularly challenging due to the absence of a universal installation method for Python packages. This issue is further compounded by backward compatibility problems where older version of codebase requires an older version of dependency (e.g., a specific version of `numpy`), making it even harder to setup old codebases. Skipping these difficult environments, on the other hand, might cause distribution bias that hurts dataset's effectiveness.

We manually configure the dependencies for each version based on the relevant configuration files (e.g., `requirements.txt`), continuous integration scripts or documentation provided in the repository. We then run the execution-based validation script from SWE-Bench to verify that golden patches (i.e., the code diff submitted by human developers) pass additional unit tests. We estimate that the entire process takes around 200 human annotation hours and 10k CPU core hours. After validation, we obtain 2,479 unit-test-validated instances from 11 different repositories. We publicly release the pre-built docker images for each instance to allow full reproducibility. Each image takes on average 2.6 GB and the image for the entire training set takes up approximately 6.4 TB.

2.2. SWE-Gym Lite

Solving software engineering tasks is computationally intensive, costing usually \$1 or more per task with frontier models (Wang et al., 2024c). To improve research efficiency, Jimenez et al. (2024) introduced SWE-Bench Lite, a canonical subset of 300 instances from SWE-Bench. Following SWE-Bench Lite filtering pipeline[4], we developed the **SWE-Gym Lite** split, comprising 230 instances. Similar to SWE-Bench Lite, this subset excludes tasks that requires editing more than one file, with poorly descriptive problem statements, excessively complex ground-truth code diffs, and tests focused on error message validation.

2.3. Dataset Statistics

We present the distribution of our dataset in Fig. 2. It illustrates that the task distribution across repositories exhibits a long-tail pattern. Notably, tasks associated with `pandas` comprise nearly one-third of the total, whereas tasks related to `bokeh` represent a mere one percent.

The statistical analysis in Tab. 2 shows that SWE-Gym has statistics similar to SWE-Bench but features higher number of lines and files edited with gold patches. Codebases in SWE-Gym, on average, have a relatively smaller number of files in the codebase but a similar number of lines of code. A higher number of edits in the gold patch may loosely correlate with increased task complexity. As reported in §B.3, this increased difficulty is also evident from the consistently lower model performance in SWE-Gym compared to SWE-Bench. Beyond models and scaffolds over-fitting to SWE-Bench, one explanation for the increased difficulty could be our deliberate inclusion of sophisticated repositories, such as `pandas` and machine-learning libraries like `MONAI`.

3. Training LM as Agent with SWE-Gym

With SWE-Gym, we train language model agents. Our primary objective is to establish reasonable baselines for SWE-Gym using two agent scaffolds (OpenHands, Wang et al. 2024c, §3.2; Moatless Tools, Örwall 2024, §3.3) and validate the effectiveness of our dataset.

3.1. Setting

Agent Scaffolds. We experiment with both two types of agents scaffolds –general-purpose prompting or specialized workflows, demonstrated through OpenHands CodeAct (Wang et al., 2024c) and MoatlessTools (Örwall, 2024).

Policy Improvement Algorithm. As a baseline, we employ a simple policy improvement algorithm: rejection sampling fine-tuning (a.k.a. filtered behavior cloning). This method is widely used as a baseline in language model × reinforcement learning literature (Zhou et al., 2024; Snell et al., 2022). At each iteration, a policy is sampled to interact with the environment. This policy may either be a stronger teacher policy or the student policy itself. Only trajectories that achieve rewards exceeding a predefined threshold are retained. The student model is then fine-tuned to mimic these high-reward trajectories using the standard negative log-likelihood loss. In our setting, the reward threshold is set to include only successful trajectories.

Evaluation Metrics. We perform evaluation on SWE-Bench Lite and SWE-Bench Verified (Jimenez et al., 2024). We measure: (1) **Resolve Rate, RR, (%)**, the percentage of resolved problems; (2) **Empty Patch, EP, (%)**, the percentage of trajectories where none of the code in the repository is edited. We use OpenHands's remote runtime (Neubig & Wang, 2024) feature to parallelize evaluation (e.g., execute unit tests).

Training Setup. We use `Qwen-2.5-Coder-Instruct` (Hui et al., 2024a) 7B, 14B, and 32B variants as our base models. For hyper-parameters and details of the training runs, please refer to §B.2.

3.2. Training General-Purpose Prompting Agents

In this subsection, we use OpenHands (version CodeActAgent 2.1, Wang et al. 2024c;b) as our agent scaffold. OpenHands CodeActAgent uses a ReAct-style (Yao et al., 2023) general-purpose prompting *without specialized workflow* (§5.1) and relies on the underlying LM to perform action and interpret observation. It equips an LM with a bash terminal and a file editor. We disable the browser feature of OpenHands in this work. We use OpenHands's remote runtime (Neubig & Wang, 2024) feature to roll out agents in parallel (i.e., execute agent's actions) on SWE-Gym.

Trajectory Collection. We roll out 491 successful trajectories where the agent successfully solves the given task from SWE-Gym. These trajectories are sampled from two models (`gpt-4o-2024-08-06` and `claude-3-5-sonnet-20241022`) with different temperature settings. Each success trajectory, on average, has 39.9 LM completion messages (roughly 19 turns) and 18,578 tokens. Please refer to Tab. 8 for more details. While SWE-Gym contains many more tasks for trajectory collection and allows repeated sampling, our current set of 491 trajectories was primarily limited by computational budget constraints rather than the availability of tasks. Please refer to Tab. 10 and §B.3 for details on the temperature setting and the success rate in the training set.

Additional Evaluation Metrics & Settings. In addition to the metrics presented in §3.1, we include the followings for further analysis: (3) **Stuck in Loop (%)**, the percentage of trajectories where the agent stuck in a loop by repeating the exact same action for the last three turns; (4) **Avg. Turn(s)**, the average number of action-observation turns for these trajectories. Our evaluation of the trained LM is

[4]For details on its construction process, see https://www.swebench.com/lite.html

Table 3: Model performance (fine-tuned on 491 SWE-Gym-sampled trajectories) on SWE-Bench (Jimenez et al., 2024) using OpenHands (Wang et al., 2024c) as agent scaffold. We use `Qwen-2.5-Coder-Instruct` as the base model. We set temperature $t = 0$ for evaluation.

Model Size	Empty Patch (%, ↓) zero-shot	fine-tuned	Δ	Stuck in Loop (%, ↓) zero-shot	fine-tuned	Δ	Avg. Turn(s) zero-shot	fine-tuned	Δ	Resolve Rate (%, ↑) zero-shot	fine-tuned	Δ
SWE-Bench Lite (300 instances)												
7B	40.3	29.7	−10.7	47.0	31.0	−16.0	20.3	22.2	+1.9	1.0(±1.0)	10.0(±2.4)	+9.0
14B	49.7	**18.1**	−31.6	31.7	27.1	−4.6	23.2	21.4	−1.8	2.7(±1.9)	12.7(±2.3)	+10.0
32B	**27.0**	**18.1**	−8.9	**16.7**	**18.1**	+1.5	**15.5**	29.3	+13.9	**3.0**(±1.4)	**15.3**(±2.5)	**+12.3**
SWE-Bench Verified (500 instances)												
7B	45.8	33.8	−12.0	39.6	**21.0**	−18.6	21.9	35.3	+13.4	1.8(±1.1)	10.6(±2.1)	+8.8
14B	44.9	14.5	−30.4	32.1	21.3	−10.7	25.5	30.1	+4.6	4.0(±1.6)	16.4(±2.0)	+12.4
32B	**9.5**	**13.8**	+4.3	**29.4**	23.8	−5.6	24.6	31.6	+7.0	**7.0**(±1.3)	**20.6**(±2.1)	**+13.6**

bounded by either 100 interaction turns or the 32k context window length, whichever is reached first. We use sampling temperature of 0 (i.e., greedy) unless otherwise specified.

Training on SWE-Gym trajectories turns LM into effective agents to fix issues. As shown in Tab. 3, despite the base model `Qwen-2.5-Coder-Instruct-32B` only performs 3.0% and 7.0% on SWE-Bench Lite and Verified, the model fine-tuned on SWE-Gym-sampled trajectories is able to achieve consistent improvements, up to 12.3% (3.0% → 15.3%) and 13.6% (7.0% → 20.6%) absolute performance (i.e., more Github issue resolved) with the largest 32B model.

Training help reduces agent's stuck-in-loop behavior. Open-weight LMs, often suffers from stuck-in-loop, especially when prompted with general-purpose prompts (§5.1): the LM would repeat the exact same action for multiple turns and never got out of this loop. This is evident in the zero-shot stuck in loop statistics in Tab. 3, where even the largest 32B model stuck in loop 30% of the time. The fine-tuned model consistently reduces the stuck in loop rate ranging from −4.6% to −18.6% for problems in both SWE-Bench Lite and Verified, with the exception of 32B model on SWE-Bench Lite (+1.5%). The reduced stuck-in-loop rate likely encourages the agent to execute more code edits, as evident by the decreased empty patch rate across different scales (except for 32B on SWE-Bench Verified).

Performance scales with model size. Rather unsurprisingly, we see the resolve rate, the empty patch, and the stuck-in-loop rate improved consistently in both SWE-Bench Lite and Verified as we switched to bigger base models (Tab. 3).

Self-improvement is not yet working. We use the fine-tuned 32B model to rollout trajectories from SWE-Gym for 6 times and obtain 868 successful trajectories (i.e. on-policy trajectories) using temperature $t = 0.5$. We further fine-tuned the base 32B model on a mixture of 868 on-policy trajectories and the previously collected 491 off-policy trajectories. When evaluating the model on SWE-Bench Lite, we observe a performance drop from 15.3% to 8.7% compared to the model that was fine-tuned using only off-policy trajectories from GPT-4 and Claude.

We hypothesize improved results could be achieved with more advanced optimization methods, such as Proximal Policy Optimization (PPO) (Schulman et al., 2017), or by employing a stronger base model. These directions remain as promising avenues for future investigation.

3.3. Self-improvement with Specialized Workflow Agents

We then explore the opposite end of the scaffold spectrum—agents with specialized workflows—using the MoatlessTools Agent (version 0.0.2; Örwall 2024) for our experiments. Unlike OpenHands CodeAct, which offers extensive freedom in long-horizon planning, this scaffold constrains the model's action space with pre-defined workflows, effectively reducing task horizons. As demonstrated in Tab. 3 and Tab. 5, we observe that open-weight models consistently deliver better zero-shot performance when paired with the MoatlessTools scaffold.

Given improved zero-shot performance and shorter task horizon with MoatlessTools scaffold, we explore if SWE-Gym allows agents to self-improve without reliance on a strong teacher. With a limited compute budget, we conduct this experiment with only 7B and 34B models, using LoRA (Hu et al., 2022) for the 34B models for improved efficiency.

We start our experiments with ablations on 7B model and scale it to 32B. We sample the model for 30 rounds on SWE-Gym-Lite with a high temperature of 1.0, adding successful trajectories to the fine-tuning dataset. This process is repeated twice, after which the improvements are marginal.

Easy Data Bias Degrades Model Performance. During repeated sampling, similar to the observation in Brown et al. (2024) , we find that the success probability for each instance follows a long-tail distribution as shown in Fig. 3, with more instances being solved simply by increasing the number of samples. While having a broader coverage of tasks should be beneficial for training, as first observed in math reasoning (Tong et al., 2024), repeated sampling introduces a distribution bias toward easier tasks, making it suboptimal to naively train on all successful trajectories.

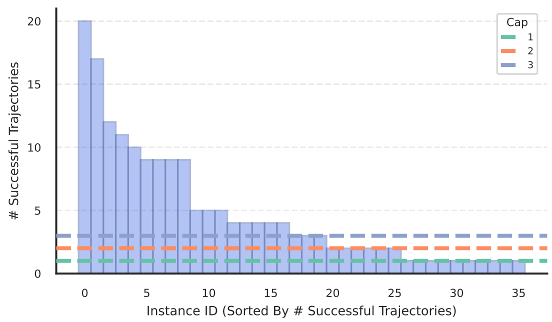

Figure 3: Success distribution over 30 rounds on SWE-Gym Lite with 7B model in zero-shot. The distribution is naturally biased toward easy tasks. Per instance capping reduces this bias but lowers the total trajectory count for training. We set temperature $t = 1$ during sampling.

Mitigate Easy Data Bias with Per Instance Capping. To mitigate such bias, Tong et al. (2024) proposes to keep the training set size same but obtain equal or more number of positive instances on difficult tasks. However, obtaining positive instances from harder tasks is inherently difficult and on average requires more rounds of sampling and therefore more compute. We consider a compute-bounded scenario with a fixed sampling budget to optimize model performance through data selection.

We use per-instance capping - a simple method limiting the maximum instances per task. As shown in Fig. 3, this approach balances dataset bias and size. Too low a cap limits dataset size and hurts performance (see §4.2), while too high a cap skews distribution toward easier tasks.

As shown in Tab. 4, empirically, we find a threshold of 2 achieves a good balance between dataset size and difficulty bias, performing marginally better than the full dataset while improving training speed. We rank trajectories by number of LM calls they make, with fewer calls preferred.

Results. With a capping of 2, we perform two rounds of policy improvements for both 7B and 32B models and present results in Tab. 5. For the 7B model, the resolve rate improves significantly, rising from 7.0% in the zero-shot setting to 9.0% after the first iteration and reaching 10.0% after the second iteration, indicating notable benefits from training on SWE-Gym. In contrast, the 32B model demonstrates strong initial performance with a zero-shot resolve rate of 19.0%, but shows only marginal improvement to 19.7% after the first iteration and no additional gains in

Cap	# Traj	Empty Patch (%, ↓)	Resolve Rate (%, ↑)
0 (Zero-shot)	0	56.3	7.0
1	36	37.3	9.0
2	62	**29**	**9.7**
3	82	43.7	7.7
No Cap (All)	172	30.7	9.3

Table 4: Resolve rate and empty patch rate on SWE-Bench Lite after 7B model trained with with data from different instance capping strategies (Cap) and therefore different number of trajectories (Traj).

Setting	7B Model		32B Model	
	EP(%, ↓)	RR(%, ↑)	EP(%, ↓)	RR(%, ↑)
Zero-Shot	56.3%	7.0%	24.3%	19.0%
Iteration 1	29.0%	9.0%	18.3%	**19.7%**
Iteration 2	**23.3%**	**10.0%**	**9.7%**	**19.7%**

Table 5: Resolve rate (RR) and Empty patch rate (EP) on SWE-Bench Lite with MoatlessTools Scaffold after online rejection sampling fine-tuning, evaluated at temperature $t = 0$. RR shown in highlighted columns.

subsequent iterations. We hypothesize that this plateau may stem from limitations inherent to the agent scaffold and the rejection sampling finetuning algorithm we employed to improve the policy.

4. Scaling Agent Improvements with SWE-Gym

4.1. Inference-Time Scaling with SWE-Gym-powered Verifier

Beyond training the agent, trajectories sampled from SWE-Gym also allow us to train a verifier (a.k.a. reward model). As a baseline, we train an outcome-supervised reward model (ORM) (Cobbe et al., 2021) that, given the relevant context of the task execution (e.g., problem statement, trajectories, git diff), generates a score representing the probability if the agent solves the problem.

We show that such learned verifiers enables effective inference-time scaling for further performance improvement. Using the same underlying LM, one can sample multiple solutions for the same problem and use the verifier to pick the best solution.

4.1.1. Verifier for Agent Scaffold with General-Purpose Prompting

For OpenHands CodeActAgent (Wang et al., 2024c;b) that uses general-purpose prompting (§5.1), we consider a verifier (ORM) setting where the input is an interleaved trajectory τ and output is a scalar reward r:

$$\tau = [o_1, a_1, o_2, a_2, \ldots, o_n, a_n], r \in [0, 1]$$

where observation o_k can be the problem-statement, command execution output, error messages, etc; action a_k can be bash command or file operations (e.g., edit, view) from the agent. We include verifier prompt template in §B.5.

Training Setup. Following the same training hyperparameters as described in §3, we train a 32B `Qwen2.5-Coder-Instruct` as verifier. We train the LM to predict 'YES' or 'NO' regarding if the agent has successfully solved the request based on the agent trajectory (see the verifier prompt in §B.5). At inference time, conditioned on the prompt and the agent trajectory τ, we use SGLang (Zheng et al., 2024a) to obtain the log probability of the next token being 'YES' (l_y) or 'NO' (l_n). We then calculate the probability of success as ORM by normalizing the log probability: $p_{\text{yes}} = \exp(l_y)/(\exp(l_y) + \exp(l_n))$.

Verifier Training Data. We re-use two sets of trajectories we sampled for agent training in §3.2: (1) **off-policy trajectories** which contain 443 successful trajectories that are sampled from `gpt-4o` and `claude-3-5-sonnet`; (2) **on-policy trajectories** which contains 875 successful trajectories sampled from the agent model (fine-tuned `Qwen2.5-Coder-Instruct-32B`). Note that we only keep trajectories that fit in the context window (32k tokens) for training. We combine both on-policy and off-policy trajectories, randomly sample the same amount of unsuccessful trajectories from each subset (1318 in total), and combine them as our dataset for verifier training.

Metrics. We measure **(1) Pass@K**, which represents the percentage of tasks where at least one successful solution is found among K sampled trajectories, and **(2) Best@K**, which selects the trajectory with the highest verification score (p_{yes}) among K samples and reports the percentage of these selected trajectories that are successful. Pass@K evaluates the model's ability to find any working solution (i.e., the upper bound for Best@K), while Best@K assesses our verifier's capability to identify the most promising solution. We calculate the mean and variance for each data point following Lightman et al. (2023), please see §B.1 for details on the calculation.

Result. Fig. 4 shows how Pass@K and Best@K metrics scale with the number of sampled agent trajectories using the fine-tuned 32B model as the agent model. Pass@K demonstrates strong improvement, rising from 20.6% to 37.8% as K increases from 1 to 8, and to 42.8% at k=16. The Best@K metric, which relies on our verifier's ability to select the best trajectory, demonstrates more modest but steady progress, improving from 20.6% at k=1 to 29.8% at k=8, and to 32.0% at k=16. The gap between Pass@K and Best@K reveals room for more advanced reward modeling techniques for coding agents. Surprisingly, we found that fine-tuning the model using LoRA (Hu et al., 2022) (29.8% @8) via Unsloth library (Unsloth Team, 2024) performs better than full-parameter fine-tuning for verifier training (27.2%@8), potentially due to the effect of regularization. Furthermore, as shown in Fig. 1 (bottom), the Best@K curve exhibits strong linearity on a logarithmic scale, indicating a promising scaling behavior.

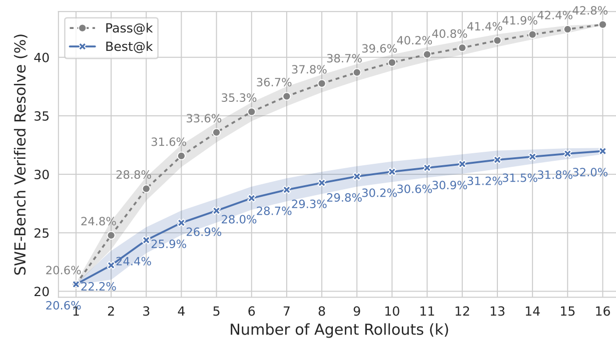

Figure 4: Scaling inference-time compute improves performance on SWE-Bench Verified using a fine-tuned verifier. Both the agent and the verifier are `Qwen2.5-Coder-Instruct-32B` model fine-tuned on the corresponding dataset (§4.1.1). OpenHands (Wang et al., 2024c) is used as the agent scaffold. The first rollout was performed with temperature $t = 0$, and $t = 0.5$ was used for the rest.

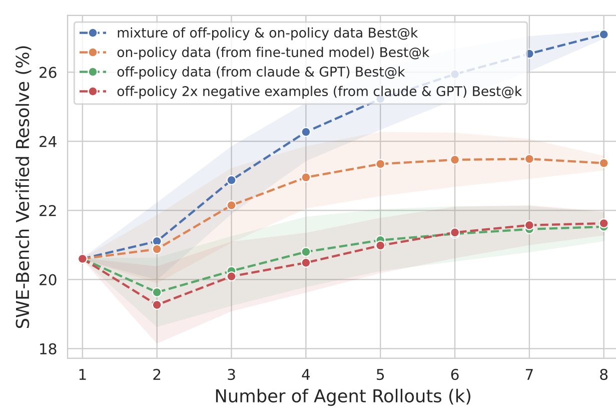

Figure 5: Abaltion study for verifier training (§4.1.1). Performances are evaluated on SWE-Bench Verified. Both the agent and the verifier are `Qwen2.5-Coder-Instruct-32B` model fine-tuned on the corresponding dataset (§4.1.1). OpenHands (Wang et al., 2024c) is used as the agent scaffold.

Ablation Study: Training Data matters for Verifier. As shown in Fig. 5, our ablation study demonstrates that the choice of training data can significantly impact verifier performance. Different from the LoRA-trained model in the previous paragraph, all ablation studies perform full-parameter fine-tuning on a 32B `Qwen2.5-Coder-Instruct` for the verifier. Training with a mixture of off-policy and on-policy data yields the best results (our default setting), with Best@k scaling from 20% to approximately 27% at k=8. In contrast, using only on-policy data from the fine-tuned model shows moderate but limited improvement, while training exclusively on off-policy data from Claude and GPT leads to early performance plateaus around 22%. Notably, using off-policy data with twice the number of negative examples performs slightly worse with a lower number of k, suggesting that overweighting negative examples during training could make the verifier overly conservative. These findings indicate that verifier training benefits most from *a diverse dataset combining both off-policy and on-policy examples*, as well as *balanced positive and negative data*, enabling better generalization and more accurate identification of successful solutions during inference time scaling.

4.1.2. Verifier for Agent Scaffold with Specialized Workflow

Figure 6: Scaling inference-time compute for MoatlessTools Agents with learned verifiers. We set temperature $t = 0.5$ during sampling.

For MoatlessTools Agents with specialized workflows, given that it doesn't have a turn-taking action-observation trajectory like OpenHands CodeActAgent, we prepare verifier inputs through a parsing process. This process combines task descriptions, relevant agent context, and generated patches, training the model to output a single confidence token indicating task success.. We adapt the context extractor from Zhang et al. (2024a) and provide the prompt template in §B.4.

We train 7B and 32B verifiers using on-policy trajectories from the last round of sampling, applying LoRA (Hu et al., 2022) for regularization. To address data bias, we cap positive data points per instance at 2 and balance the dataset by subsampling failure cases to match successful ones.

Result. We evaluate the verifiers by sampling the policy 8 times at temperature 0.5. As shown in Fig. 6, these verifiers enable effective scaling across model sizes - the 7B agent-verifier system improves from 10% to 13.3% success rate on SWE-Bench Lite, while the 32B system improves from 19.7% to 26.3%. The 7B verifier plateaus after N=4 samples when ranking trajectories from both 7B and 32B agents. In contrast, the 32B verifier continues improving even at N=8, suggesting model size affects scaling behavior.

4.2. Training-Time Scaling with Data

In this subsection, we study the impact of training data scaling on agent performance. We identify three types of data scaling for sampled trajectories: (1) **Random Scaling (No Dedup.),** which simply increases the total number of training trajectories regardless of uniqueness (Fig. 7); (2) **Unique Instance Scaling (Dedup.)**, which scales the number of distinct instances by randomly selecting one success trajectory per instance (Fig. 8); and (3) **Repository Scaling (Dedup.)**, which samples trajectories by sequentially including all instances from one repository at a time, sorted by repository name, to study the impact of repository-level diversity.

Setup. Using OpenHands (Wang et al., 2024c), we evaluate these scaling approaches on SWE-Bench Verified: random scaling on the full trajectory dataset from §3.2 (491 trajectories), unique instance scaling on these trajectories deduplicated by instance ID (294 trajectories), and repository-based scaling where we sort repositories alphabetically and include all trajectories from each repository in order (e.g., first 25% contains complete trajectories from the first N repositories). We compare models trained on 25%, 50%, and 100% of the corresponding dataset for each approach. Please refer to Tab. 7 for detailed statistics of these datasets.

Model performance scales well with more training examples. As illustrated in Fig. 7, we observe favorable scaling behavior: there is consistent improvements in model resolve rate as training data increases, particularly for the 32B model. These results suggest that SWE-Gym's current size and repository diversity are likely not performance bottlenecks - further improvements could likely be achieved by allocating more computing resources to sampling more training trajectories.

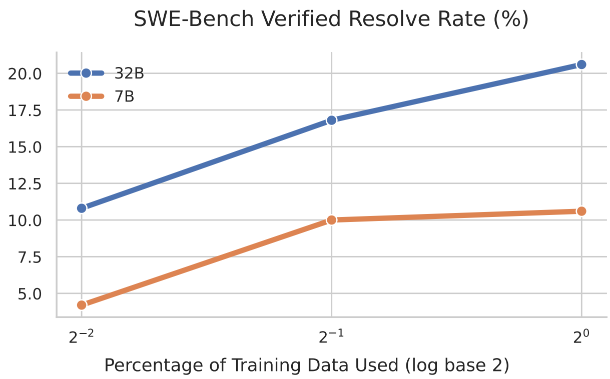

Figure 7: Model performance scaling with training data size. The x-axis shows the percentage of training data used in log base 2 scale.

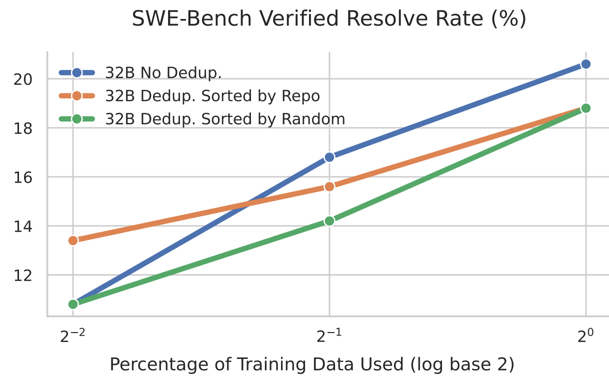

Figure 8: Comparison of three data sampling approaches: without deduplication, repository-based sampling, and random sampling (§4.2). All variants use the 32B model evaluated on SWE-Bench Verified.

Similar scaling trends suggest instance and repository diversity is not yet a bottleneck. Fig. 7 reveals comparable overall performance between different scaling approaches up to where deduplication takes effect. While Random Scaling (No Dedup.) achieves higher final performance, this is likely due to having more trajectories (491 vs 294) rather than better scaling efficiency. Among deduplicated approaches, Repository Scaling shows stronger initial performance at 25% data, suggesting that complete repository coverage may provide more coherent learning signals early in training. These results suggest that the repository and instance diversity of SWE-Gym is not yet a bottleneck - further improvements could likely be achieved by simply sampling more agent trajectory data for traning, regardless of duplication or repository distribution.

5. Related Work

5.1. Agents that Solves Github Issues

In this paper, we focus on software engineering agents designed to automatically resolve GitHub issues, specifically in the context of SWE-Bench (Jimenez et al., 2024). These agents operate in a well-defined setting: given a GitHub issue and its associated code repository, they generate a valid code modification (i.e., git diff patch) to address the issue. The correctness of these modifications is verified through the execution of a test suite written by human developers.

Existing agent designs can be categorized by the number of human priors injected into the agent workflow: **(1) specialized workflow**: Xia et al. (2024); Örwall (2024); Zhang et al. (2024b); Chen et al. (2024) build specialized workflow with human-defined stages (e.g., localization, code edit, patch re-ranking selection), where an LM is iteratively prompted to produce the final results. While restricting the LM to work on pre-defined subtasks, this approach effectively reduces the task horizon and alleviates the need for long-term planning. However, these specialized workflows face limitations: they require significant human engineering to define appropriate stages and transitions, may not generalize well to novel issue types outside their designed workflow, and can be brittle when intermediate steps fail. (2) **general-purpose prompting**: Systems such as Yang et al. (2024); Wang et al. (2024c) rely on general prompting approaches, including methods like ReAct (Yao et al., 2023) and CodeAct (Wang et al., 2024b). These systems primarily depend on the LM itself to plan through long horizons and produce the next action based on an evolving history of actions and observations, without enforcing a fixed workflow or relying heavily on human pre-defined stages. While more flexible, these general approaches place higher demands on the underlying LM's capabilities and can be computationally expensive due to multiple rounds of interaction with a growing interaction history.

Most existing successful methods are based on prompting proprietary models and crafting specialized workflow to workaround existing limitations of those models. This starkly contrasts the success of learning-based approaches in other domains (Silver et al., 2017; Akkaya et al., 2019), where a system learns from prior interactions and rewards to develop task competency progressively. We argue that proper training environments and baselines can help accelerate research in this direction. We verify that we can use SWE-Gym to build strong agents through learning.

5.2. Environments for Training Software Agents

There is no existing dataset suitable for training software engineering agents. Existing datasets often exhibit distinct constraints. SWE-Bench (Jimenez et al., 2024) is widely used for evaluating software engineering performance. The training split lacks executable environments and success signals present in the evaluation split, making its usefulness limited for model training. HumanEval (Chen et al., 2021) is designed for standalone code generation tasks, akin to coding competitions. Therefore, it falls short of addressing the complex challenges inherent in real-world, repository-level software engineering tasks, which involve thousands of files, millions of lines of code, and tasks such as bug fixing, feature development, and system optimization. Similarly, R2E (Jain et al., 2024) is a small evaluation dataset with 246 instances and, due to its synthetic nature, lacks the realism and complexity in real world software engineering scenario. There is also limited empirical evidence if this environment is useful for training. Our proposed SWE-Gym instead uses real-world GitHub issue bodies as task instructions, and uses the associated unit tests for validation. This approach introduces a realistic and complex task formulations, aligning closely with real-world software engineering challenges.

5.3. Post-training: From Chatbots, Reasoners to Agents

Post-training, which finetunes a pre-trained language model, usually through supervised or reinforcement learning, has proven highly effective in improving model performance across various domains. RLHF (Ouyang et al., 2022) has become a standard method for adapting language models into chatbots, improving both performance and alignment of the model (Team, 2024). In mathematical reasoning, two standard datasets, MATH (Hendrycks et al., 2021b) and GSM-8K (Cobbe et al., 2021), have question and answer pairs for both training and evaluation, which enables researchers to explore methods to train both policy, and verifiers (reward models) (Cobbe et al., 2021; Wang et al., 2024a).

Earlier works (Wang et al., 2024b; Chen et al., 2023; Zeng et al., 2023; Wu et al., 2024) shows distilling agent trajectories from stronger models consistently improves weaker models. Recent works shift towards self-improving methods. Xi et al. (2024); Zhai et al. (2024); Bai et al. (2024) show how RL or rejection sampling fine-tuning, guided by a reward signal, can enable models to self-improve without reliance on a more capable teacher model.

Despite the wide success, post-training generally rely on either expert demonstration data or a training environment with reliable reward signals, which is largely missing in the software engineering domain. This limitation has contributed to a reliance on prompting-based methods with proprietary language models, as reflected in most top-performing approaches on SWE-bench (Jimenez et al., 2024). Our work addresses this gap with SWE-Gym, a training environment grounded in real-world software engineering tasks, using associated expert written tests as the reward signal. Our experiments verify that we can use SWE-Gym to build strong SWE agents without any prompt engineering. Concurrent with our work, Ma et al. (2024) and Golubev et al. (2024) both study the training of software engineering agents, with the latter also exploring verifiers training and dataset construction. We discuss the key differences between these concurrent works in §A.

6. Conclusions

In this paper, we introduce SWE-Gym, the first open training environment that bridges critical gaps in enabling scalable learning for software engineering agents. By combining real-world Python tasks with repository-level context, pre-configured execution environments, and test verifications, SWE-Gym will be a foundation for advancing LM agent training research. Through extensive experiments, we demonstrate that SWE-Gym enables both agent and verifier models to achieve significant improvements in resolving complex software tasks. Our findings highlight the scalability of these approaches, revealing potential for continuous performance gains with increased compute.

7. Acknowledgments

We thank John Yang, Ofir Press for helpful discussions. John Yang for assistance in reproducing data analysis results from SWE-Bench. We thank Modal Labs[5] for the GPU compute support through its Academic Credits Program.

[5] https://modal.com/

References

Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019.

Badertdinov, I., Trofimova, M., Anapolskiy, Y., Abramov, S., Zainullina, K., Golubev, A., Polezhaev, S., Litvintseva, D., Karasik, S., Fisin, F., Skvortsov, S., Nekrashevich, M., Shevtsov, A., and Yangel, B. Scaling data collection for training software engineering agents. *Nebius blog*, 2024.

Bai, H., Zhou, Y., Cemri, M., Pan, J., Suhr, A., Levine, S., and Kumar, A. Digirl: Training in-the-wild device-control agents with autonomous reinforcement learning. *ArXiv*, abs/2406.11896, 2024. URL https://api.semanticscholar.org/CorpusID:270562229.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., R'e, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling. *ArXiv*, abs/2407.21787, 2024. URL https://api.semanticscholar.org/CorpusID:271571035.

Chen, B., Shu, C., Shareghi, E., Collier, N., Narasimhan, K., and Yao, S. Fireact: Toward language agent fine-tuning. *ArXiv*, abs/2310.05915, 2023. URL https://api.semanticscholar.org/CorpusID:263829338.

Chen, D., Lin, S., Zeng, M., Zan, D., Wang, J.-G., Cheshkov, A., Sun, J., Yu, H., Dong, G., Aliev, A., Wang, J., Cheng, X., Liang, G., Ma, Y., Bian, P., Xie, T., and Wang, Q. Coder: Issue resolving with multi-agent and task graphs. *CoRR in ArXiv*, abs/2406.01304, 2024.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pondé, H., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D. W., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Babuschkin, I., Balaji, S., Jain, S., Carr, A., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M. M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021. URL https://api.semanticscholar.org/CorpusID:235755472.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *ArXiv*, abs/2110.14168, 2021. URL https://api.semanticscholar.org/CorpusID:239998651.

Golubev, A., Polezhaev, S., Zainullina, K., Trofimova, M., Badertdinov, I., Anapolskiy, Y., Litvintseva, D., Karasik, S., Fisin, F., Skvortsov, S., Nekrashevich, M., Shevtsov, A., Abramov, S., and Yangel, B. Leveraging training and search for better software engineering agents. *Nebius blog*, 2024. https://nebius.com/blog/posts/training-and-search-for-software-engineering-agents.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and Steinhardt, J. Measuring coding challenge competence with APPS. In Vanschoren, J. and Yeung, S. (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021a. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c24cd76e1ce41366a4bbe8a49b02a028-Abstract-round2.html.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D. X., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *ArXiv*, abs/2103.03874, 2021b. URL https://api.semanticscholar.org/CorpusID:232134851.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024a.

Hui, B., Yang, J., Cui, Z., Yang, J., Liu, D., Zhang, L., Liu, T., Zhang, J., Yu, B., Dang, K., et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024b.

Jain, N., Shetty, M., Zhang, T., Han, K., Sen, K., and Stoica, I. R2E: turning any github repository into a programming agent environment. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=kXHgEYFyf3.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. R. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step. *ArXiv*, abs/2305.20050, 2023. URL https://api.semanticscholar.org/CorpusID:258987659.

Ma, Y., Cao, R., Cao, Y., Zhang, Y., Chen, J., Liu, Y., Liu, Y., Li, B., Huang, F., and Li, Y. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

Modal. Modal: High-performance AI infrastructure. https://modal.com/, 2024. Accessed: 2024-12-18.

Neubig, G. and Wang, X. Evaluation of LLMs as Coding Agents on SWE-Bench (at 30x Speed!), 2024. URL https://www.all-hands.dev/blog/evaluation-of-llms-as-coding-agents-on-swe-bench-at-30x-speed.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Pan, J., Zhang, Y., Tomlin, N., Zhou, Y., Levine, S., and Suhr, A. Autonomous evaluation and refinement of digital agents. *ArXiv*, abs/2404.06474, 2024. URL https://api.semanticscholar.org/CorpusID:269009430.

PyTorch Team. torchtune: PyTorch native post-training library. https://github.com/pytorch/torchtune, 2024.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *ArXiv*, abs/1707.06347, 2017. URL https://api.semanticscholar.org/CorpusID:28695052.

Shao, Z., Wang, P., Zhu, Q., Xu, R., Song, J., Bi, X., Zhang, H., Zhang, M., Li, Y., Wu, Y., et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *ArXiv*, abs/1712.01815, 2017. URL https://api.semanticscholar.org/CorpusID:33081038.

Snell, C. B., Kostrikov, I., Su, Y., Yang, M., and Levine, S. Offline rl for natural language generation with implicit language q learning. *ArXiv*, abs/2206.11871, 2022. URL https://api.semanticscholar.org/CorpusID:249954054.

Tao, N., Ventresque, A., Nallur, V., and Saber, T. Enhancing program synthesis with large language models using many-objective grammar-guided genetic programming. *Algorithms*, 17(7):287, 2024. doi: 10.3390/A17070287. URL https://doi.org/10.3390/a17070287.

Team, Q. Qwen2.5: A party of foundation models, September 2024. URL https://qwenlm.github.io/blog/qwen2.5/.

Tong, Y., Zhang, X., Wang, R., Wu, R. M., and He, J. Dart-math: Difficulty-aware rejection tuning for mathematical problem-solving. *ArXiv*, abs/2407.13690, 2024. URL https://api.semanticscholar.org/CorpusID:271270574.

Unsloth Team. Easily finetune and train LLMs. Get faster with unsloth. https://unsloth.ai/, 2024.

Wang, P., Li, L., Shao, Z., Xu, R., Dai, D., Li, Y., Chen, D., Wu, Y., and Sui, Z. Math-shepherd: Verify and reinforce LLMs step-by-step without human annotations. In Ku, L.-W., Martins, A., and Srikumar, V. (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 9426–9439, Bangkok, Thailand, August 2024a. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-long.510. URL https://aclanthology.org/2024.acl-long.510.

Wang, X., Chen, Y., Yuan, L., Zhang, Y., Li, Y., Peng, H., and Ji, H. Executable code actions elicit better LLM agents. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024b. URL https://openreview.net/forum?id=jJ9BoXAfFa.

Wang, X., Li, B., Song, Y., Xu, F. F., Tang, X., Zhuge, M., Pan, J., Song, Y., Li, B., Singh, J., Tran, H. H., Li, F., Ma, R., Zheng, M., Qian, B., Shao, Y., Muennighoff, N., Zhang, Y., Hui, B., Lin, J., Brennan, R., Peng, H., Ji, H., and Neubig, G. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *CoRR in ArXiv*, abs/2407.16741, 2024c.

Wu, Z., Bai, H., Zhang, A., Gu, J., Vinod Vydiswaran, V., Jaitly, N., and Zhang, Y. Divide-or-conquer? which part should you distill your llm? *ArXiv*, 2024.

Xi, Z., Ding, Y., Chen, W., Hong, B., Guo, H., Wang, J., Yang, D., Liao, C., Guo, X., He, W., Gao, S., Chen, L., Zheng, R., Zou, Y., Gui, T., Zhang, Q., Qiu, X., Huang, X., Wu, Z., and Jiang, Y.-G. Agentgym: Evolving large language model-based agents across diverse environments. *ArXiv*, abs/2406.04151, 2024. URL https://api.semanticscholar.org/CorpusID:270285866.

Xia, C. S., Deng, Y., Dunn, S., and Zhang, L. Agentless: Demystifying llm-based software engineering agents. *CoRR*, abs/2407.01489, 2024. doi: 10.48550/ARXIV.2407.01489. URL https://doi.org/10.48550/arXiv.2407.01489.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. Swe-agent: Agent-computer interfaces enable automated software engineering. *CoRR*, abs/2405.15793, 2024. doi: 10.48550/ARXIV.2405.15793. URL https://doi.org/10.48550/arXiv.2405.15793.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

Yuan, L., Cui, G., Wang, H., Ding, N., Wang, X., Deng, J., Shan, B., Chen, H., Xie, R., Lin, Y., Liu, Z., Zhou, B., Peng, H., Liu, Z., and Sun, M. Advancing LLM reasoning generalists with preference trees. *CoRR*, abs/2404.02078, 2024. doi: 10.48550/ARXIV.2404.02078. URL https://doi.org/10.48550/arXiv.2404.02078.

Zeng, A., Liu, M., Lu, R., Wang, B., Liu, X., Dong, Y., and Tang, J. Agenttuning: Enabling generalized agent abilities for llms. In *Annual Meeting of the Association for Computational Linguistics*, 2023. URL https://api.semanticscholar.org/CorpusID:264306101.

Zhai, Y., Bai, H., Lin, Z., Pan, J., Tong, S., Zhou, Y., Suhr, A., Xie, S., LeCun, Y., Ma, Y., and Levine, S. Fine-tuning large vision-language models as decision-making agents via reinforcement learning. *ArXiv*, abs/2405.10292, 2024. URL https://api.semanticscholar.org/CorpusID:269790773.

Zhang, K., Yao, W., Liu, Z., Feng, Y., Liu, Z., Murthy, R., Lan, T., Li, L., Lou, R., Xu, J., Pang, B., Zhou, Y., Heinecke, S., Savarese, S., Wang, H., and Xiong, C. Diversity empowers intelligence: Integrating expertise of software engineering agents. *ArXiv*, abs/2408.07060, 2024a. URL https://api.semanticscholar.org/CorpusID:271860093.

Zhang, Y., Ruan, H., Fan, Z., and Roychoudhury, A. Autocoderover: Autonomous program improvement. In *ISSTA*, 2024b.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024a. URL https://arxiv.org/abs/2312.07104.

Zheng, T., Zhang, G., Shen, T., Liu, X., Lin, B. Y., Fu, J., Chen, W., and Yue, X. Opencodeinterpreter: Integrating code generation with execution and refinement. *ArXiv*, abs/2402.14658, 2024b. URL https://api.semanticscholar.org/CorpusID:267782452.

Zhou, Y., Zanette, A., Pan, J., Levine, S., and Kumar, A. Archer: Training language model agents via hierarchical multi-turn rl. *ArXiv*, abs/2402.19446, 2024. URL https://api.semanticscholar.org/CorpusID:268091206.

Örwall, A. Moatless Tool. https://github.com/aorwall/moatless-tools, 2024. Accessed: 2024-10-22.

A. Comparison with Concurrent Works

Ma et al. (2024) trains an LM agent, Lingma SWE-GPT, using a method similar to our rejection sampling fine-tuning baseline, with a dataset comparable to our SWE-Gym Raw splits. Without executable unit test feedback, they rely on manually defined heuristics to filter out low-quality trajectories, such as comparing similarity between submitted patches and edit locations with gold patches. The model weights are publicly accessible but not the training pipeline or the dataset.

Most relevant to our work are two consecutive blog posts by Golubev et al. (2024) and Badertdinov et al. (2024), who also construct an executable training environment with real-world tasks from GitHub. Instead of manual configuration, they employ a general environment setup script and simply discard instances that fail the setup process. This approach leads to key differences in dataset size and distribution: while it biases the environment away from tasks with complex dependencies, they successfully collect 6,415 instances, about 1.5 times larger than our dataset. In Golubev et al. (2024), they also study training agents and verifiers with the environment. Additionally, they explore a lookahead setting where a trained verifier ranks and selects the best next action. With a substantially large collection of agent trajectories (80,036 compared to thousands in our experiments) and model size (72B compared to 32B), Their best system achieves 40% accuracy on SWE-Bench Verified. While their dataset and agent trajectories are publicly accessible, the model is not.

In comparison, with a comparable dataset size, our SWE-Gym has executable feedback, avoids potential dataset bias through manual configuration of environments, while providing comprehensive analysis of agent and verifier training, their scaling behaviors, and positive results on agent self-improvement. Our system achieves competitive results with significantly lower compute and a smaller model size (32B vs 72B). Lastly, we open source all artifacts of the project, including dataset, model weights, agent trajectory data and the training pipeline.

Model	**SWE-Bench**		**Openness**	
Name, Model Size	Lite	Verified	Model	Environment
Ma et al. (2024), 72B	22.0	30.2	✓	✗
Golubev et al. (2024) Agent and Verifier, 72B	-	40.6	✗	✓
Our SWE-Gym Agent and Verifier, 32B	26.0	32.0	✓	✓

Table 6: Comparison of model performance on SWE-Bench benchmark and if the model weights and environments are publically accessible (openness).

B. Experiment Details

B.1. Mean and Variance for Pass@N and Best@N.

We mostly follow (Lightman et al., 2023) for obtaining the mean and variance for the Pass@N and Best@N curve. Given a total of M rounds of rollouts, for $N < M$, we calculate the mean and variance across 100 randomly selected sub-samples of size N from the M rollouts. For the OpenHands CodeActAgent inference-time scaling curve at §4, we exclude this calculation for N=1 , as we use a temperature of 0 for the first attempt.

B.2. Training Details.

OpenHands Agent Experiment. We use `torchtune` (PyTorch Team, 2024) for full parameter fine-tuning with a learning rate of `1e-4`, maximum 5 epochs, global batch size of 8, max context length of `32768`. We fine-tuned both 7B, 14B, and 32B variant of the model, and experiments were performed with 2-8x NVIDIA H100 80G GPU on modal (Modal, 2024). The only exception is in the main experiment of §4.1.1, where we use LoRA (Hu et al., 2022) (29.8% @8) via Unsloth library (Unsloth Team, 2024) to train the verifier for max 2 epochs, while other hyper-parameter stays the same.

MoatlessTools Agent Experiment. All MoatlessTools models are trained with a context window of `10240`. For experiments with the 7B model, we use torchtune to train the policy model with full-finetuning using 4 H100 GPUs. We set batch size to 8, learning rate to 2×10^{-5}, and train for 5 epochs.

For the 32B model, we use Unsloth (Unsloth Team, 2024) with a single H100 GPU for LoRA fine-tuning. We set the number of epochs to 5, batch size to 8, LoRA rank to 64, and learning rate to 5×10^{-4}. We use the same configuration for verifier training.

Table 7: Distribution of success trajectories used in training-time scaling experiments (§4.2). **Dedup.** denotes that the trajectories are deduplicated by randomly select ONE success trajectory per instance ID; **Sorted by random (repo) X% (Dedup.)** denotes a subset of trajectories taken from the first X% from dedup. instances that are sorted randomly (by repository name).

	Original	**Dedup.**	**Sorted by Random (Dedup.)**		**Sorted by Repo (Dedup.)**	
			First 25%	First 50%	First 25%	First 50%
getmoto/moto	155	72	12	33	0	46
Project-MONAI/MONAI	95	53	17	25	53	53
pandas-dev/pandas	70	61	14	30	0	0
python/mypy	46	27	7	12	0	0
dask/dask	45	29	8	17	6	29
iterative/dvc	36	24	8	12	0	0
conan-io/conan	20	12	1	7	12	12
pydantic/pydantic	11	7	2	4	0	0
facebookresearch/hydra	7	5	2	5	0	5
bokeh/bokeh	3	2	1	1	2	2
modin-project/modin	3	2	1	1	0	0
Total	491	294	73	147	73	147

Table 8: Statistics of SWE-Gym-sampled trajectories. We use the tokenizer from `Qwen-2.5-Coder-Instruct-7B` to estimate the number of tokens.

	Resolved	**Count**	**Mean**	**Std**	**Min**	**Max**	**Percentiles** 5%	10%	25%	50%	75%	90%	95%
Num. of Messages	✗	5,557.0	39.2	31.9	7.0	101.0	9.0	9.0	9.0	29.0	61.0	100.0	101.0
	✓	491.0	39.9	19.9	13.0	101.0	19.0	21.0	25.0	33.0	47.5	65.0	87.0
Num. of Tokens	✗	5,557.0	17,218.3	17,761.6	1,615.0	167,834.0	1,833.0	1,907.0	2,268.0	12,305.0	26,434.0	41,182.2	51,780.6
	✓	491.0	18,578.5	11,361.4	2,560.0	81,245.0	5,813.0	8,357.0	11,559.5	15,999.0	22,040.5	31,632.0	39,512.5

Agent	Model	Model Size	Training Data	Resolved (%)
		SWE-Bench Verified (500 instances)		
RAG	SWE-Llama (Jimenez et al., 2024)	7B	10K instances	1.4
RAG	SWE-Llama (Jimenez et al., 2024)	13B	10K instances	1.2
Lingma Agent (Ma et al., 2024)	Lingma SWE-GPT (v0925)	7B	90K PRs from 4K repos	18.2
Lingma Agent (Ma et al., 2024)	Lingma SWE-GPT (v0925)	72B	90K PRs from 4K repos	28.8
OpenHands (Wang et al., 2024c) (Ours)	fine-tuned Qwen2.5-Coder-Instruct	32B	491 agent trajectories from 11 repos	20.6
OpenHands w/ Verifier (Wang et al., 2024c) (Ours)	fine-tuned Qwen2.5-Coder-Instruct	32B (Agent & Verifier)	491 agent trajectories from 11 repos for agent + 1318 × 2 success/failure agent trajectories for verifier	**32.0**

Table 9: Performance comparison with SWE-Bench (Jimenez et al., 2024) baselines *with publicly accessible weights*. Data source: `https://www.swebench.com/`, Accessed on Dec 21, 2024.

Table 10: Summary of trajectories sampled from SWE-Gym.

Trajectory Set	Sampled from Model	Sampled on Dataset	Temperature	Max Turns	Success trajectories
D_0	`gpt-4o-2024-08-06`	SWE-Gym Lite	0	30	19 (8.26%)
			(Cumulative) Total D_0		**19**
$D_1 \setminus D_0$	`gpt-4o-2024-08-06`	SWE-Gym Lite	0.2	30	11 (4.78%)
	`gpt-4o-2024-08-06`	SWE-Gym Lite	0.3	30	17 (7.39%)
	`gpt-4o-2024-08-06`	SWE-Gym Lite	0.4	30	21 (9.13%)
	`gpt-4o-2024-08-06`	SWE-Gym Lite	0.5	30	18 (7.83%)
	`gpt-4o-2024-08-06`	SWE-Gym Lite	0.8	30	20 (8.70%)
			(Cumulative) Total D_1		**106**
$D_2 \setminus D_1$	`gpt-4o-2024-08-06`	SWE-Gym Lite	0	50	19 (8.26%)
	`claude-3-5-sonnet-20241022`	SWE-Gym Lite	0	50	67 (29.1%)
	`gpt-4o-2024-08-06`	SWE-Gym Full	0	50	*111 (4.55%)
	`gpt-4o-2024-08-06`	SWE-Gym Full	1	50	188 (7.71%)
			(Cumulative) Total D_2		**491**

* Run into infrastructure-related error where some instances failed to complete, this number might be under estimate of actual number of success trajectories.

For MoatlessAgent experiments, we serve the agent with FP8 quantization for improved throughput, which we found to have minimal effects on model performance.

B.3. Details of OpenHands Trajectory Sampling

As detailed in Tab. 10, we collect a few sets of trajectories for fine-tuning experiments. We collect dataset D_0 by sample `gpt-4o-2024-08-06` on SWE-Gym Lite with temperature 0 and collected 19 trajectories that eventually solve the task (evaluated by unit test in SWE-Gym). We then varied the temperatures (setting `t={0.2, 0.3, 0.4, 0.5, 0.8}`) and sample on SWE-Gym Lite. Combining these instances with D_0, we get 106 trajectories that solve the given problem (D_1). We set the maximum number of turns to be 30 for both D_0 and D_1. To experiment on the effect of max turn, we set max number of turns to 50 and sample `gpt-4o-2024-08-06` (19 resolved out of 230) and `claude-3-5-sonnet-20241022` (67 resolved out of 230) with temperature 0 on SWE-Gym Lite, and sample `gpt-4o-2024-08-06` (temperature `t={0, 1}`) on SWE-Gym full set (in total 299 resolved out of 4876 instances). This gives us in in total 106 + 19 + 67 + 299 = 491 success trajectories, which forms our final training trajectories D_2.

B.4. MoatlessTools ORM Prompt

The following is a pseudo-code that generates a prompt for MoatlessTools Verifier (ORM), which is modified from (Zhang et al., 2024a). Unlike (Zhang et al., 2024a), which relies on proprietary models like Claude-3.5-Sonnet for context extraction, we obtain context directly from the agent's trajectory being evaluated.

```
SYSTEM_MESSAGE = """You are an expert in python for software engineering and code
↪  review. Your responsibility is to review the patches generated by language
↪  models to fix some issues and provide feedback on the quality of their
↪  code."""

USER_MESSAGE="""I want you to evaluate an LLM-generated candidate patch that
↪  tries to resolve an issue in a codebase.

To assist you in this task, you are provided with the following information:
 - You are given an issue text on a github repository (wrapped with
  ↪  <issue_description></issue_description>).
 - You are also given some identified code spans that are relevant to the issue.
    Each code span is wrapped with <code_span file_path=FILE_PATH
     ↪  span_id=SPAN_ID></code_span> tags, where FILE_PATH is the path to the
     ↪  file containing the code span, and SPAN_ID is the unique identifier for
     ↪  the code span.
    Each code span also comes with the line numbers for you to better understand
     ↪  the context. It's possible that the code span are not sufficient to fix
     ↪  the issue, adjust your score accordingly.
 - You are given the candidate patch that tries to resolve the target issue.
    For your convenience, you are given the hunks of original code and the code
     ↪  after applying the patch.
    The code before the patch is wrapped with <before_patch></before_patch> and
     ↪  the code after the patch is wrapped with <after_patch></after_patch>.
    Note that the file names in before_patch starts with 'a/' and the file names
     ↪  in after_patch starts with 'b/'.


<issue_description>
{issue_text}
</issue_description>

<before_patch>
{before_patch}
</before_patch>

<after_patch>
{after_patch}
</after_patch>

{code_spans}

Response in "True" or "False" for whether the patch has resolved the issue."""
```

B.5. OpenHands ORM Prompt

The following is a pseudo-code that generates a prompt for OpenHands Verifier (ORM).

```
SYSTEM_MESSAGE = '''You are an expert judge evaluating AI assistant interactions.
↪  Your task is to determine if the assistant successfully resolved the user's
↪  request.

Key evaluation criteria:
1. Did the assistant complete the main task requested by the user?
```

```
2. Did the assistant handle all edge cases and requirements specified?
3. Were there any errors or issues in the final solution?
4. Did the assistant verify the solution works as intended?

Respond only with "<judgement>YES</judgement>" or "<judgement>NO</judgement>".'''

USER_MESSAGE = '''Please evaluate the following interaction between an AI
↪  assistant and a user:

=== INTERACTION LOG ===
''' + traj_str + '''
=== END INTERACTION ===

Based on the above interaction, did the assistant successfully resolve the user's
↪  initial request? Respond with YES or NO.'''

messages = [
    {'role': 'system', 'content': SYSTEM_MESSAGE},
    {'role': 'user', 'content': USER_MESSAGE},
    {'role': 'assistant', 'content': '<judgement>' + ("YES" if resolved else
     ↪  "NO") + '</judgement>'}
]
```

The last assistant messages that contains judgement is only provided during training time. At inference time, the trained verifier is responsible predicting the probability of ‘Yes’ and ‘No’.