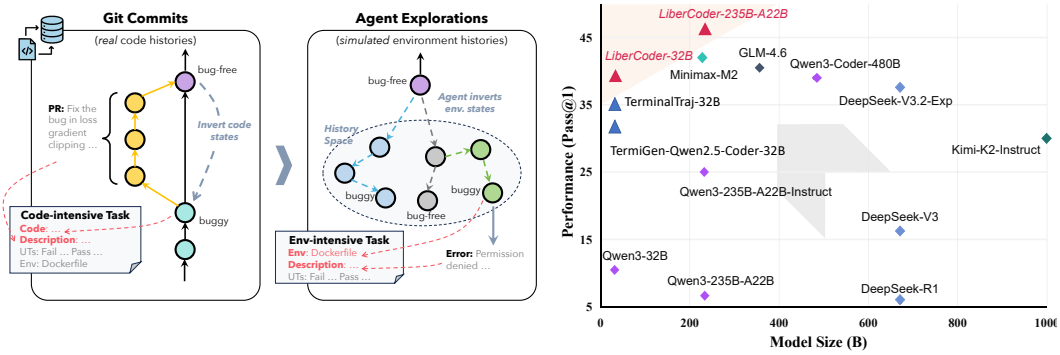
CLI-GYM: SCALABLE CLI TASK GENERATION VIA AGENTIC ENVIRONMENT INVERSION

Anonymous authors
Paper under double-blind review



(a) Deriving tasks by tracing code/environment histories (b) Pass@1 on Terminal-Bench 1.0 vs model size

Figure 1: Overview of CLI-Gym and downstream results on Terminal-Bench 1.0. **(a):** Code-centric tasks can use repository histories such as pull requests, whereas CLI-Gym starts from unit-test-verified healthy repository environments and uses execution-feedback-guided agents to construct reproducible degraded states. **(b):** Fine-tuning Qwen3-32B and Qwen3-235B-A22B-Instruct on the 291 filtered CLI-Gym trajectories yields LiberCoder models with pass@1 scores of 38.9% and 46.1%, respectively, under the OpenHands scaffold. ▲ and ⋄ denote methods using open and closed data pipelines; cross-paper points may use different scaffolds and inference budgets.

ABSTRACT

Command-line-interface (CLI) agents must diagnose and repair dependencies, permissions, system files, runtime configurations, and source code, yet executable training data for these interactions remain limited. We introduce CLI-Gym, an execution-feedback-guided agentic environment inversion pipeline. Starting from healthy, unit-test-verified repository environments, an inversion agent constructs counterfactual degraded states, summarizes the changes as reproducible Dockerfiles, and pairs observed failures with natural-language issues and executable verifiers. From 4,066 inversion prompts, CLI-Gym produces 1,655 verified CLI task instances; 81.0% contain an environment modification. A repair model solves 417 tasks, from which shortcut and length filtering retains 291 trajectories for supervised fine-tuning. Under the same OpenHands scaffold, these trajectories improve Qwen3-235B-A22B-Instruct by 21.1 points to 46.1% pass@1 on Terminal-Bench 1.0 and by 12.9 points to 31.0% on Terminal-Bench 2.0, with consistent gains at the 32B scale. A three-expert stratified audit separately characterizes the realism, solvability, issue–test consistency, and shortcut resistance of the full task collection. These results establish the training value of the filtered teacher-solvable subset while making the quality boundary of the complete collection explicit.

1 INTRODUCTION

Interacting with a runtime environment is a central part of software development. In addition to editing source code, developers diagnose dependencies, permissions, system files, environment variables, services, and runtime configuration. We use *CLI task* as the broad term for tasks completed

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Table 1: Representative pipelines for constructing executable agentic coding tasks. The methods use different source environments, construction procedures, and output types, so instance counts are contextual rather than a controlled quality comparison.

Method	Gold Instance ⟶	Data Engine ⟶	Problem Instance	Collection	# Instances
Code-Intensive Task					
SWE-Bench (Jimenez et al., 2024)		PR-based		Auto	2294
SWE-Gym (Pan et al., 2024)		PR-based	Code-Intensive Issue	Auto	2438
R2E-Gym (Jain et al., 2025)	Pre-installed Github Repo	PR-based / LLM-based	+	Auto	8135
SWE-smith (Yang et al., 2025b)		PRs / LLM Synthesis	Environment / Failed Unit Tests	Auto	50137
SWE-Dev (Du et al., 2025)		Test-Driven		Auto	14000
Environment-Intensive Task					
Terminal-Bench 1.0 (Merrill et al., 2026)	-	-	CLI Issue	Human-written	80
Terminal-Bench 2.0 (Merrill et al., 2026)	-	-	+	Human-written	89
CLI-Gym (ours)	Pre-installed Github Repo	Agentic Synthesis	Environment / Failed Unit Tests	Auto	1655

through command-line interaction. Within this space, an *environment-only* task changes dependencies, system files, permissions, environment variables, or configuration outside the repository source tree; a *source-only* task changes only repository source files; and a *mixed* task involves both. This operational taxonomy is deliberately not an absolute division of capabilities: practical CLI tasks jointly exercise environment diagnosis, tool use, source-code reasoning, and long-horizon control.

Executable and verifiable environments have enabled rapid progress on code-centric agent training, where repository histories provide commits, pull requests, issues, and tests from which defective states can be reconstructed Jimenez et al. (2024); Jain et al. (2025); Yang et al. (2025b). Environment histories are less directly observable: two developers may share a commit while differing in installed packages, system configuration, permissions, or external services. Recent work, including Terminal-Traj and TermiGen, has begun to address open CLI trajectory and task generation Wu et al. (2026); Zhu et al. (2026), but the construction routes differ. This paper focuses on a complementary route: actively degrading a healthy repository environment under execution feedback and preserving the resulting state and verifier.

CLI-Gym begins with a repository snapshot whose unit tests pass. An inversion agent then executes changes inside an isolated container, observes command and test feedback, and searches for a reproducible degraded state. Rather than claiming to recover a real historical environment, this procedure constructs a plausible counterfactual history. The agent summarizes the degradation in a Dockerfile; failing tests provide an executable verifier; and an issue generator describes the observed symptoms. This execution-feedback loop distinguishes CLI-Gym from single-pass issue or Dockerfile synthesis.

The resulting data have distinct stages and evidence boundaries. From 4,066 inversion prompts over 29 curated Python repositories, the pipeline assembles 1,655 verified issue–environment–verifier tuples. Of these tasks, 417 (25.2%) are solved by the repair rollout model, and 291 (17.6%) remain after filtering short or shortcut-based trajectories. The supervised fine-tuning (SFT) experiments therefore evaluate the training value of the 291 filtered, teacher-solvable trajectories; they do not establish uniform quality for all 1,655 tasks. We assess the full collection separately with a stratified audit by three non-author experts. Operationally, 48.1% of tasks are environment-only, 19.0% source-only, and 32.9% mixed, so 81.0% contain an environment modification.

Under a fixed OpenHands scaffold, CLI-Gym supervision yields consistent improvements at two Qwen3 model scales and on Terminal-Bench 1.0 and 2.0. For Qwen3-235B-A22B-Instruct, the combined training setting improves pass@1 by 21.1 points to 46.1% on version 1.0 and by 12.9 points to 31.0% on version 2.0. Cross-paper leaderboard results are reported only as context because backbones, scaffolds, and inference budgets differ.

Our contributions are threefold: (1) we introduce execution-feedback-guided agentic environment inversion, which constructs reproducible CLI tasks from healthy, unit-test-verified repository environments; (2) we build 1,655 automatically verified task instances and characterize their composition and quality with an operational taxonomy and a three-expert audit; and (3) we show that 291 filtered trajectories provide data-efficient CLI supervision, with consistent gains across two model scales and two benchmark versions.

2 Related Work

CLI Agents and Benchmarks. Tool-integrated agents such as Claude Code, Gemini CLI, Codex CLI, and OpenHands interact with software through shell commands, file operations, and execution feedback Anthropic (2025); Google (2025); OpenAI (2025); Wang et al. (2025). Terminal-Bench evaluates this setting with containerized tasks that include software engineering, system administration, security, and other terminal workflows Terminal-Bench Team (2025); Merrill et al. (2026). Performance depends on the joint choice of backbone, agent scaffold, tool interface, context budget, and sampling policy; consequently, leaderboard entries with different settings are not controlled comparisons of training data.

Scalable Environments for Code-Centric Agents. SWE-bench derives executable software-repair tasks from real pull requests Jimenez et al. (2024). SWE-Gym, R2E-Gym, SWE-Smith, and SWE-Dev scale related environments through repository mining, hybrid verifiers, synthetic faults, and test-driven feature development Pan et al. (2024); Jain et al. (2025); Yang et al. (2025b); Du et al. (2025). These pipelines benefit from version-controlled code histories and repository-native tests. CLI-Gym adopts the same principle of executable verification but targets a broader state that includes dependencies, system configuration, permissions, environment variables, and runtime artifacts.

CLI Data Generation and Trajectory Collection. TerminalTraj collects large-scale trajectories from Dockerized terminal environments, while TermiGen synthesizes terminal environments and repair trajectories Wu et al. (2026); Zhu et al. (2026). CLI-Gym differs in its starting point and artifact: it begins from a unit-test-verified healthy repository environment, uses an agent with execution feedback to induce a failure, and records the degradation as a replayable Dockerfile paired with an issue and verifier. The approaches are complementary, and their reported downstream scores use different models and scaffolds. We therefore compare their construction choices qualitatively and use cross-paper scores only as descriptive context.

3 Method

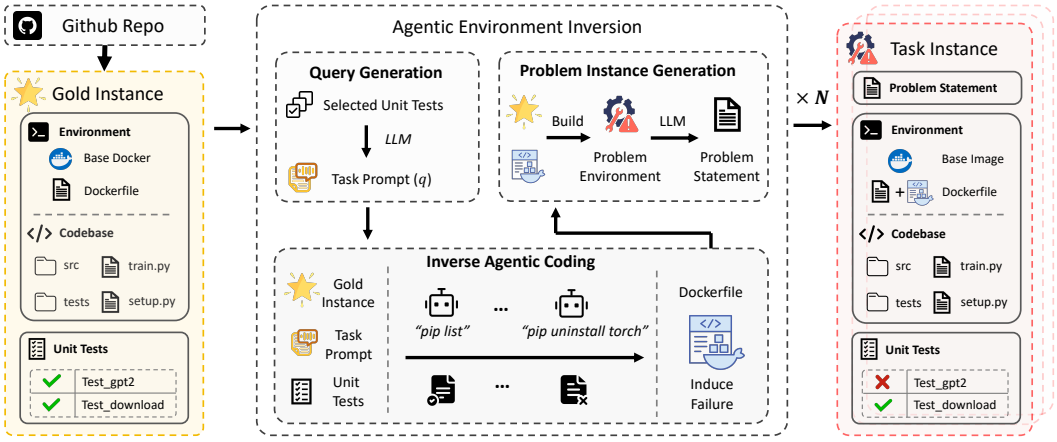


Figure 2: Overview of our proposed CLI-Gym pipeline. 1) Starting from a GitHub repository, we construct a gold instance consisting of a functional environment, codebase, and associated unit tests. 2) We then derive task prompts from the unit tests and execute them with an agent to obtain failure-inducing commands. Based on the observed execution commands and failing tests, we automatically generate a corresponding problem statement. 3) Finally, the outputs from the previous steps are assembled into a standardized task instance.

The central premise of this work is that *agentic coding* can be viewed as a process in which an autonomous agent modifies the state of its execution environment through code edits and environment operations. In conventional agentic coding tasks, the agent is typically required to transform an initially defective environment into a correct one, *i.e.*, a transition from a *poor* state to a *gold* state

Table 2: CLI-Gym generation and training-data funnel. Percentages for the last three stages use the 1,655 verified tasks as the denominator.

Stage	Count	Share
Inversion prompts	4,066	–
Verified task instances	1,655	100.0%
Rollout-solved tasks	417	25.2%
Filtered SFT trajectories	291	17.6%

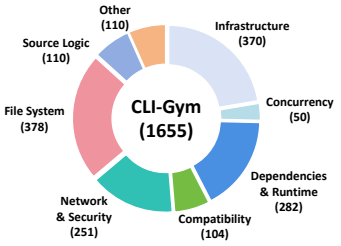


Figure 3: Category distribution of problem instances we generated using CLI-Gym.

(Sec. 3.1). In this paper, we have an inverse perspective to reinterpret task collection as agentic environment inversion, where an agent deliberately degrades a gold environment into a poor one (Sec. 3.2) to simulate environment histories. This perspective enables automated synthesis of CLI task instances and repair trajectories (Sec. 3.3).

3.1 Agentic Coding from an Environment Perspective

To establish a clear understanding of our approach, we first formalize agentic coding from an environment-centric perspective, treating the execution environment, rather than code alone, as the primary object of manipulation. We begin by formalizing the state of a coding environment as

$$\mathcal{S} = (\mathcal{B}, \mathcal{D}, \mathcal{C}, \mathcal{R}), \tag{1}$$

where $\mathcal{B}$ denotes the base image; $\mathcal{D}$ denotes the reproducible build instructions, including dependency and system-level changes; $\mathcal{C}$ denotes the repository source tree; and $\mathcal{R}$ denotes build-time and runtime state such as installed packages, system files, permissions, environment variables, and service configuration. Under this representation, agentic coding can be modeled as a state transition:

$$\mathcal{S}_{\text{poor}} \xrightarrow[\text{Agent}]{\Delta\mathcal{S}} \mathcal{S}_{\text{gold}}, \tag{2}$$

where $\mathcal{S}_{\text{poor}}$ is a defective state and $\mathcal{S}_{\text{gold}}$ is a state in which the repository's selected unit tests pass. The action sequence $\Delta\mathcal{S}$ may edit source code or modify the broader environment. Changes made interactively are summarized into build instructions so that the final state can be replayed.

3.2 Instance Construction by Environment Inversion

As suggested by Equation 2, scaling agentic coding training instances fundamentally requires access to a large and diverse set of initial poor states $\mathcal{S}_{\text{poor}}$. Motivated by this observation, we formulate data collection as an agentic environment inversion process:

$$(\mathcal{S}_{\text{gold}}, \mathcal{T}_{\text{passed}}) \xrightarrow[\text{Agent}]{\Delta\mathcal{S}^{-}} (\mathcal{S}_{\text{poor}}, \mathcal{T}_{\text{failed}}), \tag{3}$$

where $\mathcal{T}_{\text{passed}}$ and $\mathcal{T}_{\text{failed}}$ denote test outcomes before and after degradation. Starting from a *gold* state, an autonomous agent applies a failure-inducing action sequence $\Delta\mathcal{S}^{-}$ and observes command and test feedback. This process constructs a counterfactual degraded state rather than recovering an environment that necessarily existed in repository history. A state becomes a task instance only after the pipeline assembles a replayable environment, natural-language issue, and executable verifier.

Initial Gold Environment Construction. Our gold instances are Docker images derived from pinned snapshots of 29 Python repositories curated from the SWE-Smith repository set (Yang et al., 2025b). Repository selection and environment construction require human engineering; the subsequent per-task generation pipeline is automated. We install each repository from its base image and retain the instance only after the configured unit-test suite passes. This healthy, unit-test-verified image serves as $\mathcal{S}_{\text{gold}}$ for subsequent inversion. Repository statistics are listed in Table 13; the data release will record the corresponding commits and licenses.

Environment Inversion by Agentic Dockerfile Coding. As shown in Figure 2, we start from a gold environment and its passing unit tests. CLI-Gym uses one base degradation-generation template, samples one to three directions from five intervention directions, samples up to 200 candidate unit tests, and includes summaries of previous tasks to reduce repetition. A second refinement template is applied stochastically to improve disruption–test alignment. After inversion, one of three repair-issue templates with different guidance strengths is sampled. The full templates are provided in the appendix.

```
FROM task-pandas:latest  # Gold environment

RUN mkdir -p /tmp/corrupted
RUN cp /opt/.../lib/libsqlite3.so /tmp/corrupted/
RUN cp /opt/.../lib/libz.so /tmp/corrupted/

# Corrupt ELF headers to break dynamic linking
RUN dd if=/dev/zero of=/tmp/corrupted/libsqlite3.so
    \
      bs=1 count=24 seek=8 conv=notrunc
RUN dd if=/dev/zero of=/tmp/corrupted/libz.so \
      bs=1 count=24 seek=8 conv=notrunc

# Replace libraries with corrupted versions
RUN cp /tmp/corrupted/* /opt/.../lib/
```

Figure 4: A simplified Dockerfile that corrupts shared libraries in a healthy pandas environment, inducing failures that require system-level diagnosis.

DeepSeek-V3.2 (Liu et al., 2025) performs inversion-prompt generation, environment inversion through OpenHands, repair-issue generation, and repair rollout. Direct prompt and issue generation use temperature 1.0; OpenHands inversion and rollout use greedy decoding with temperature 0. Formally, given a gold environment $\mathcal{S}_{\text{gold}}$ and task prompt q, the agent manipulates the filesystem, dependencies, permissions, configuration, and source tree, producing

$$\mathcal{S}_{\text{poor}} = \mathcal{S}_{\text{gold}} \oplus \Delta\mathcal{S}^{-}, \tag{4}$$

where $\Delta\mathcal{S}^{-}$ is the failure-inducing action sequence. The resulting Dockerfile encodes the final degradation so that it can be replayed. Figure 4 shows an environment-level example involving shared-library integrity. The yield is generator-dependent: on the same 512 pre-generated prompts, Qwen3-Coder-30B-A3B-Instruct produces 63 successful environments (12.3%), whereas DeepSeek-V3.2 produces 131 (25.6%). Thus, the inversion agent is replaceable, but the pipeline is not generator-independent.

Execution-based Problem Instance Construction. Given the synthesized Dockerfile, we rebuild the environment from the original gold state and run the selected tests. If at least one test executes and fails, the observed failures become *fail-to-pass* cases and the remaining successful tests become *pass-to-pass* cases. If the test runner cannot execute, the pipeline records a runtime/test-infrastructure failure for bookkeeping; this does not imply that every selected functionality is broken. If all tests execute and pass, generation is unsuccessful and the candidate is discarded. The issue generator uses the degradation prompt and observed failure output to create a natural-language report. A verified task therefore contains (i) a replayable environment, (ii) a natural-language issue, and (iii) an executable verifier.

This procedure produces automatically verified task instances, while realism, solvability, issue–test consistency, and shortcut resistance require separate quality evidence.

3.3 CLI-Gym

Generation Funnel and Composition. We apply CLI-Gym to 29 curated Python repositories. As summarized in Table 2, 4,066 inversion prompts yield 1,655 verified task instances. Based on the targets of the final inversion operations, 796 tasks are environment-only (48.1%), 315 are source-only (19.0%), and 544 are mixed (32.9%); hence, 81.0% contain an environment modification. This operational classification complements the semantic category distribution in Figure 3 and prevents treating every instance as a pure environment task.

Expert Audit. Automatic execution checks establish that a task has a replayable degraded environment and a verifier, but they do not by themselves establish task quality. We therefore ask three non-author experts to independently evaluate stratified samples from the generated, rollout-solved, and filtered stages. *Realism* measures whether the failure resembles a plausible maintenance incident; *solvability* asks whether the supplied issue and environment admit a feasible repair; *issue–test consistency* measures whether the verifier captures the reported problem; and *shortcut resistance* checks whether the task can be bypassed through unintended artifacts or trivial workarounds. Raters are not told the CLI-Gym funnel stage, borderline cases default to fail, and overall inter-rater agreement is substantial (Fleiss' $\kappa \approx 0.72$, $p < 0.001$).

Table 3: Point estimates from the stratified three-expert audit of CLI-Gym stages. Rates average binary expert–example judgments. Small cross-stage differences should not be interpreted as controlled effects.

Stage	Audited	Realism	Solvability	Consistency	Shortcut resistance
Generated tasks	160	70.0%	59.8%	80.0%	79.8%
Rollout-solved tasks	40	81.7%	98.3%	96.7%	78.3%
Filtered tasks	30	84.4%	97.8%	95.6%	88.9%

As shown in Table 3, the generated collection has relatively strong issue–test consistency and shortcut resistance, but its 59.8% solvability and 70.0% realism reveal a quality gap that executable verification alone does not detect. The rollout-solved sample reaches 98.3% solvability and 96.7% consistency, while the final filtered sample reaches 88.9% shortcut resistance and retains similarly high solvability and consistency. These results support viewing CLI-Gym as a quality funnel: the 1,655 generated tasks provide broad coverage with heterogeneous quality, whereas the 291 filtered tasks form a smaller, higher-confidence training subset. Because the rows are separately sampled stage populations rather than a controlled ablation, the shifts are descriptive and should not be assigned to any single filtering operation. The complete protocol, confidence intervals, item-level judgments, and descriptive reference results for TerminalTraj and Terminal-Bench are reported in Section A.5 and the anonymous artifact.

Trajectory Collection with CLI-Gym. To demonstrate the effectiveness of CLI-Gym, we leverage the generated instances to collect agent trajectories for model training. Using DeepSeek-V3.2 as the repair rollout model, we solve 417 of 1,655 tasks (25.2%). We remove trajectories that exploit shortcuts or unintended solutions and those with fewer than 20 interaction steps, retaining 291 trajectories (17.6% of the verified collection) for SFT. These stages introduce selection bias: downstream training results establish the utility of the filtered teacher-solvable subset, whereas evidence about the full collection comes from the expert audit. In the experiments, *Raw-Success Traj.* denotes the 417 rollout-solved trajectories and *Filtered-Success Traj.* denotes the 291 retained trajectories.

4 Experiments

We train language models on the filtered trajectories collected with CLI-Gym and evaluate them with a fixed OpenHands scaffold on Terminal-Bench 1.0 and 2.0.

4.1 Experimental Setups

Generation and Agent Stack. DeepSeek-V3.2 serves all generation and rollout roles: inversion-prompt generation, environment inversion, repair-issue generation, and repair rollout. Direct prompt and issue-generation calls use temperature 1.0; environment inversion and repair rollout use temperature 0 through OpenHands 1.1.0. The execution harness is based on Terminal-Bench 0.2.0. We use OpenHands throughout generation and evaluation to hold the scaffold and tool interface fixed; results should not be interpreted as scaffold-independent.

Training. We fine-tune Qwen3-32B and Qwen3-235B-A22B-Instruct using a two-stage procedure. The first stage uses 48K open-source software-engineering trajectories to initialize general repository navigation, tool use, and program-repair behavior. The second stage uses the 291 Filtered-Success Traj. Optimization details are documented in the appendix; the data release will document trajectory sources, licenses, and preprocessing.

Evaluation. We evaluate agents on Terminal-Bench 1.0 and 2.0 (Terminal-Bench Team, 2025), two widely adopted benchmarks for assessing agentic interaction with real-world terminal environments. Terminal-Bench comprises collections of environment manipulation and system-level tasks, 80 in v1.0 and 89 in v2.0, where agents must interact with a Linux environment via command-line interfaces to diagnose and resolve realistic software and system issues.

Table 4: Pass@1 on **Terminal-Bench 1.0** and **Terminal-Bench 2.0**. We report pass@1 scores for **LiberCoder-32B** and **LiberCoder-235B-A22B** evaluated using the OpenHands agent framework. Results for other models are taken from the official Terminal-Bench leaderboards. Rows outside the OpenHands block may use different agent scaffolds and inference budgets and are included only for context. Models marked with † are evaluated by us.

Model	Open weights	Terminal-Bench 1.0	Terminal-Bench 2.0
Performance with OpenHands			
Claude Haiku 4.5 Anthropic (2025e)	✗	-	13.9
Gemini 2.5 Pro Comanici et al. (2025)	✗	-	16.4
Grok 4 xAI (2025)	✗	-	27.2
Claude Sonnet 4 Anthropic (2025a)	✗	41.3	-
Claude Opus 4.1 Anthropic (2025b)	✗	-	36.9
Claude Sonnet 4.5 Anthropic (2025d)	✗	42.7†	42.6
GPT-5 Singh et al. (2025)	✗	-	43.8
Claude Opus 4.5 Anthropic (2025c)	✗	-	51.9
Qwen3-32B (Yang et al., 2025a)	✓	10.3†	5.7†
Qwen3-235B-A22B-Instruct (Yang et al., 2025a)	✓	25.0†	18.1†
Qwen3-Coder-30B-A3B-Instruct (Yang et al., 2025a)	✓	26.5†	12.9†
Qwen3-Coder-480B-A35B-Instruct (Yang et al., 2025a)	✓	-	25.4
Kimi-K2-Instruct (Team et al., 2025)	✓	-	**26.7**
LiberCoder-32B	✓	**38.9**	19.5
LiberCoder-235B-A22B	✓	**46.1**	**31.0**
Performance with Any Agent and Closed-source Data Pipeline			
Gemini 2.5 Pro Comanici et al. (2025)	✗	25.3	32.6
Grok 4 xAI (2025)	✗	39.0	27.2
Claude Haiku 4.5 Anthropic (2025e)	✗	41.8	29.8
Claude Opus 4.1 Anthropic (2025b)	✗	43.8	38.0
Claude Sonnet 4.5 Anthropic (2025d)	✗	51.0	42.8
Claude Opus 4.5 Anthropic (2025c)	✗	-	57.8
GPT 5.2 OpenAI (2025)	✗	-	62.9
Gemini 3 Pro Google DeepMind (2025)	✗	-	64.7
GPT-OSS-120B Agarwal et al. (2025)	✓	-	18.7
Kimi-K2-Instruct Team et al. (2025)	✓	30.0	27.8
Qwen3-Coder-30B-A3B-Instruct Yang et al. (2025a)	✓	31.3	12.9†
Qwen3-Coder-480B-A35B-Instruct Yang et al. (2025a)	✓	39.0	27.2
GLM-4.6 Z.ai (2025)	✓	40.5	24.5
Minimax-M2 MiniMax (2025a)	✓	42.0	30.0
Minimax-M2.1 MiniMax (2025b)	✓	-	36.6
Performance with Any Agent and Open-source Data Pipeline			
OpenThinker-Agent-v1 Team (2025)	✓	-	4.9
TerminalTraj-7B Wu et al. (2026)	✓	23.0	10.1
TermiGen-Qwen3-32B Zhu et al. (2026)	✓	27.5	-
TerminalTraj-14B Wu et al. (2026)	✓	28.9	19.1
TermiGen-Qwen2.5-Coder-32B Zhu et al. (2026)	✓	31.3	-
TerminalTraj-32B Wu et al. (2026)	✓	35.3	**22.0**
LiberCoder-32B	✓	**38.9**	19.5
LiberCoder-235B-A22B	✓	**46.1**	**31.0**

Following the Terminal-Bench protocol, a task is solved when its verification scripts pass. Pass@1 and pass@3 are obtained from separate experiments: pass@1 reports one standalone run per task, whereas pass@3 reruns each task three times and counts a task as solved if at least one attempt succeeds. Against Terminal-Bench 1.0 and 2.0 jointly, our data have one repository-name match and zero matches in commit SHA, Docker image digest, base image, unit-test path, or test content. The same-name repository uses different commits, images, tests, and tasks. These checks rule out the measured forms of direct instance overlap, not possible exposure of public repositories during model pretraining.

4.2 Main Results

Table 4 reports LiberCoder and representative leaderboard entries on Terminal-Bench 1.0 and 2.0. The controlled comparison is between each LiberCoder model and its corresponding base model under the same OpenHands scaffold; other entries provide context only.

Table 5: Ablation study on the effects of open-source agentic trajectories and our generated trajectories across model scales.

SWE Traj.	Filtered-Success Traj.	Pass@1	Pass@3
Qwen3-32B			
		10.3	*19.1*
✓		22.1 (+11.8)	28.1 (+9.0)
	✓	32.4 (+22.1)	37.9 (+18.8)
✓	✓	**38.9 (+28.6)**	**44.5 (+25.4)**
Qwen3-235B-A22B-Instruct			
		25.0	*32.4*
✓		28.6 (+3.6)	34.5 (+2.1)
	✓	39.5 (+14.5)	46.2 (+13.8)
✓	✓	**46.1 (+21.1)**	**53.9 (+21.5)**

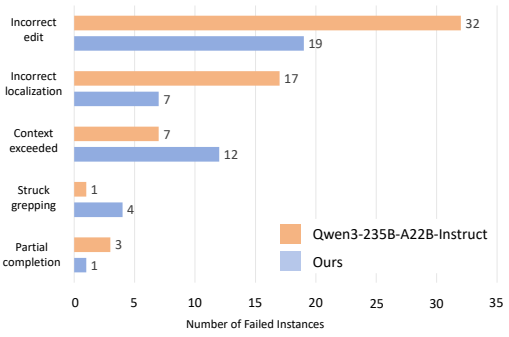


Figure 5: Failure-type distribution on Terminal-Bench 1.0 for LiberCoder-235B-A22B and Qwen3-235B-A22B-Instruct.

Table 6: We ablate the effects of trajectory filtering by comparing models trained on filtered and raw successful trajectories, with and without extra open-source agentic trajectories, on Qwen3-32B.

SWE Traj.	Raw-Success Traj.	Filtered-Success Traj.	Pass@1
		✓	32.4
	✓		33.8
✓	✓		36.4
✓		✓	**38.9**

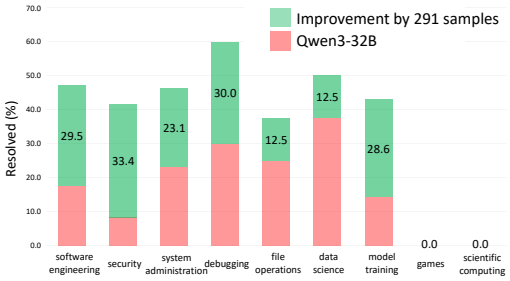


Figure 6: Category-wise pass@3 on Terminal-Bench 1.0 for Qwen3-32B before and after CLI-Gym training.

With SFT on 291 filtered trajectories after SWE initialization, LiberCoder-32B and LiberCoder-235B-A22B improve over their respective base models by 28.6 and 21.1 points on Terminal-Bench 1.0 and by 13.8 and 12.9 points on Terminal-Bench 2.0. The direction of improvement is consistent across both model scales and benchmark versions. Because the training set is selected by teacher success and filtering, this result demonstrates the utility of that subset rather than the uniform quality of all 1,655 tasks.

LiberCoder-32B reaches 38.9 on Terminal-Bench 1.0 and LiberCoder-235B-A22B reaches 46.1 and 31.0 on versions 1.0 and 2.0, respectively. These scores are competitive with open-weight leaderboard entries trained by other open data pipelines. Since those systems differ in backbone, scaffold, and inference budget, we do not use the cross-paper ranking as evidence of universal superiority.

4.3 Ablation Studies

SWE and CLI-Gym Supervision. Table 5 compares generic SWE trajectories with the 291 CLI-Gym trajectories. At both model scales, CLI-Gym-only training produces a larger gain than 48K SWE trajectories, while their combination performs best. This supports the narrower conclusion that CLI interaction supervision is scarce in the generic SWE data and complementary to it. The experiment does not isolate the individual contributions of environment diagnosis, tool use, and long-horizon control.

Environment Diversity via Scaling Repositories. Different repositories induce distinct system configurations, resulting in diverse failure modes and recovery dynamics. To isolate the effect of repository diversity, we perform an ablation in which the total number of training trajectories is fixed at 100, while varying the number of source repositories used for data generation.

As shown in Figure 7, performance improves as more repositories are included under this fixed 100-trajectory setting. Within the tested repositories, this result associates source-environment diversity with better transfer; it does not establish coverage beyond the 29 Python repositories.

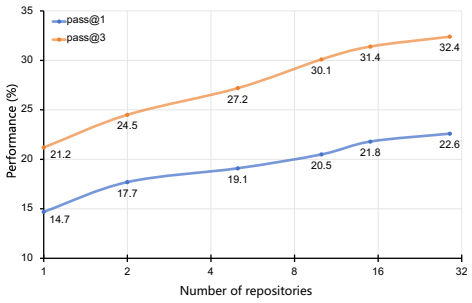


Figure 7: Effect of environment diversity under a fixed data budget. We vary the number of source repositories while keeping the total number of 100 CLI-Gym trajectories fixed.

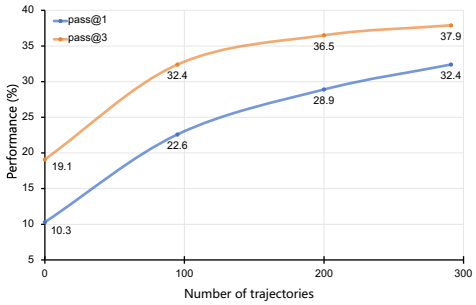


Figure 8: Effect of trajectory scaling on Terminal-Bench 1.0 performance. We fine-tune Qwen3-32B using different proportions of the 291 Filtered-Success trajectories.

Filtering Low-Quality Trajectories. Table 6 analyzes trajectory filtering under different training settings. Without SWE pretraining, using all 417 rollout-solved trajectories slightly outperforms the 291 filtered subset. With SWE initialization, the filtered subset performs better despite containing fewer trajectories. Thus, the effect of filtering depends on model initialization; the result does not imply that every discarded task or trajectory is intrinsically low quality.

Data Scaling. Figure 8 examines progressively larger subsets of the filtered CLI-Gym trajectories while holding the SWE trajectories fixed. Performance improves and then plateaus beyond approximately 200 trajectories in this setting, motivating further work on task diversity and coverage.

4.4 Results and Failure Case Visualization

Category-wise Improvement. Figure 6 illustrates category-wise changes after training with CLI-Gym. Gains are concentrated in software engineering, system administration, security, and debugging, while gaming and scientific-computing tasks remain challenging. The category results are descriptive because category sizes are limited and CLI tasks often combine multiple capabilities.

Failure Type Distribution. Figure 5 presents the distribution of failure types on Terminal-Bench 1.0 for Qwen3-235B-A22B-Instruct before and after training. The trained model shows fewer editing and localization failures but explores for longer in some cases, increasing the frequency of exceeding the 128k inference context limit.

5 Conclusion

In this paper, we present CLI-Gym , an execution-feedback-guided environment inversion pipeline for constructing CLI tasks. The pipeline represents degraded states with replayable Dockerfiles and pairs them with issues and executable verifiers. Based on this toolkit, we assemble 1,655 verified task instances and select 291 filtered trajectories for SFT. Experiments show that this selected supervision improves CLI-agent performance, leading to competitive Terminal-Bench performance under OpenHands. More precisely, CLI-Gym constructs 1,655 verified CLI tasks through execution-feedback-guided inversion, while SFT uses 291 filtered teacher-solvable trajectories. The consistent gains at two model scales and on two benchmark versions demonstrate the value of this selected supervision. The expert audit and operational task taxonomy characterize the broader collection without assuming uniform quality or pure environment focus. CLI-Gym is therefore a step toward open, reproducible CLI-agent data construction rather than a resolution of task quality, domain coverage, or scaffold-independent generalization.

AI Use Statement

Generative AI is an object of study and an implementation component in this work: DeepSeek-V3.2 generates inversion prompts and repair issues and acts through OpenHands during environment inversion and repair rollout. For paper writing, generative AI tools were used only for language

polishing and grammar correction. We reviewed the generated tasks, experimental code, analyses, and manuscript text, verified reported results against executable artifacts.

Reproducibility Statement

The method section specifies the task tuple and generation funnel, while the appendix documents prompt templates, tool interfaces, framework versions, decoding parameters, filtering, training, checkpoint selection, pass@k aggregation, overlap checks, and repository statistics. The anonymous audit artifact at `https://anonymous.4open.science/r/CLI-Gym-Audit-8D8E/` provides the rubric, sampling manifests, item-level judgments, bootstrap confidence intervals, and reproduction scripts. The planned data release includes task Dockerfiles, verifiers, trajectories, pinned repository commits, licenses, and configuration files.

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Anthropic. Introducing claude 4. `https://www.anthropic.com/news/claude-4`, 2025a. Accessed: 2026-01-20.

Anthropic. Introducing claude 4.1 opus. `https://www.anthropic.com/news/claude-opus-4-1`, December 2025b. Accessed: 2026-01-27.

Anthropic. Introducing claude 4.5 opus. `https://www.anthropic.com/news/claude-opus-4-5`, November 2025c. Accessed: 2026-01-27.

Anthropic. Introducing claude sonnet 4.5. `https://www.anthropic.com/news/claude-sonnet-4-5`, September 2025d. Accessed: 2026-01-26.

Anthropic. Introducing claude haiku 4.5. `https://www.anthropic.com/news/claude-haiku-4-5`, October 2025e. Accessed: 2026-01-26.

Anthropic. Claude Code. `https://github.com/anthropics/claude-code`, 2025. Accessed: 2026-01-26.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975*, 2025.

Google. Gemini CLI. `https://github.com/google-gemini/gemini-cli`, 2025. Accessed: 2026-01-26.

Google DeepMind. A new era of intelligence with gemini 3. `https://blog.google/products-and-platforms/products/gemini/gemini-3/`, November 2025. Accessed: 2026-01-27.

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

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 (ICLR)*, 2024.

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

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

MiniMax. Minimax m2: An efficient model for the agentic era. https://www.minimax.io/news/minimax-m2, October 2025a. Accessed: 2026-01-27.

MiniMax. Minimax m2.1: Significantly enhanced multi-language programming, built for real-world complex tasks. https://www.minimax.io/news/minimax-m21, December 2025b. Accessed: 2026-01-27.

OpenAI. Codex. https://github.com/openai/codex, 2025. Accessed: 2026-01-26.

OpenAI. Introducing gpt-5.2. https://openai.com/index/introducing-gpt-5-2/, December 2025. Accessed: 2026-01-27.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. *arXiv preprint arXiv:2412.21139*, 2024.

Aaditya Singh, Adam Fry, Adam Perelman, Adam Tart, Adi Ganesh, Ahmed El-Kishky, Aidan McLaughlin, Aiden Low, AJ Ostrow, Akhila Ananthram, et al. Openai gpt-5 system card. *arXiv preprint arXiv:2601.03267*, 2025.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

OpenThoughts-Agent Team. OpenThoughts-Agent. https://www.open-thoughts.ai/blog/agent, December 2025.

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

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, et al. The openhands software agent sdk: A composable and extensible foundation for production agents. *arXiv preprint arXiv:2511.03690*, 2025.

Siwei Wu, Yizhi Li, Yuyang Song, Wei Zhang, Yang Wang, Riza Batista-Navarro, Xian Yang, Mingjie Tang, Bryan Dai, Jian Yang, et al. Large-scale terminal agentic trajectory generation from dockerized environments. *arXiv preprint arXiv:2602.01244*, 2026.

xAI. Grok 4. https://x.ai/news/grok-4, July 2025. Accessed: 2026-01-27.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

John Yang, Kilian Lieret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. SWE-smith: Scaling data for software engineering agents. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track (NeurIPS)*, 2025b.

Z.ai. Glm-4.6. https://z.ai/blog/glm-4.6, 2025. Accessed: 2026-01-28.

Kaijie Zhu, Yuzhou Nie, Yijiang Li, Yiming Huang, Jialian Wu, Jiang Liu, Ximeng Sun, Zhenfei Yin, Lun Wang, Zicheng Liu, et al. Termigen: High-fidelity environment and robust trajectory synthesis for terminal agents. *arXiv preprint arXiv:2602.07274*, 2026.

A Detailed CLI-Gym Pipeline

This section presents the full technical details of the CLI-Gym pipeline. We construct gold instances from 29 different open-source GitHub repositories. A gold instance consists of the environment, codebase, and unit tests. Based on these gold instances, we further generate task instances. The repository information, the number of unit tests per repository, and the final number of task instances produced are summarized in Table 13.

A.1 Query Generation

After constructing a repository as a gold instance, we first randomly sample 1–3 intervention directions and randomly select 200 unit tests from the repository, which are injected into a prompt with previous task titles and provided to LLM to obtain an initial task prompt. The specific prompt is shown in Figure 11.

Once a task prompt in a predefined format is generated, we randomly apply a second-stage prompt refinement to make the task more closely aligned with the selected unit tests, improving both yield and diversity. The refinement prompt used in our pipeline is shown in Figure 12.

After generating a task prompt specification, we embed it into a task template (Figure 13) and package it as an agent task. Each task consists of: (1) a Docker-compose.yaml file for mapping trajectories, logs, and other artifacts between the executing container and local path; (2) a Dockerfile specifying the base image; (3) a run-tests.sh script containing the unit tests execution commands; and (4) a task.yaml file that provides the concrete task prompt.

A.2 Environment Inversion

We execute agentic tasks using a modified Terminal-Bench harness. Specifically, docker-compose.yaml is used to mount trajectory files and logs into the container and to invoke the Dockerfile, which specifies the base image and launches the agent. The agent is then prompted with the task prompt that specifies how an agent should deliberately induce failures and allow any operation to interact with the environment and finish the task. Each task is executed for approximately 15 minutes and is terminated when the agent outputs a final_thought. A sample trajectory is shown in Figure 14.

After task execution, the harness automatically runs run-tests.sh inside the container to evaluate the selected unit tests. Based on the test outcomes, we apply the following rules:

- If some unit tests fail, the failed tests are recorded as fail-to-pass tests, while the successful ones are recorded as pass-to-pass tests.
- If the test command fails to execute, the candidate is accepted as a valid runtime/test-infrastructure failure: the inability to launch the verifier itself demonstrates that the inversion has damaged the execution environment. All selected unit tests remain in the verification set, but no individual test is labeled as fail-to-pass or pass-to-pass because no per-test outcome is observed.
- If all unit tests pass, the task is considered unsuccessful and discarded.

In addition, the harness prompts the agent to summarize a Dockerfile that captures the degraded environment, enabling deterministic reproduction of the failure that the agent built. A complete Dockerfile demo is shown in Figure 21.

A.3 Task Generation

After the inverse task is completed, we generate a corresponding problem statement targeting the induced faults. The problem statement is constructed from the original task prompt together with the failed unit tests information. Concretely, we randomly select one of three prompts with different levels of guidance: the prompt shown in Figure 15 asks the LLM to generate a more explicit and strongly guided issue description, while the prompt shown in Figure 16 encourages a weaker, less directive issue formulation, and the prompt shown in Figure 17 balances direction and difficulty.

Regardless of which prompt is used, the LLM is required to output a hint. This hint can optionally be removed by a rule-based filter, allowing us to derive two distinct repair tasks (with or without hints) from the same issue.

The generated statement is then inserted into the problem task template (Figure 18). Together with the extracted unit tests and Dockerfile, this forms a complete repair task.

All run-tests.sh templates used in our pipeline are provided in Figure 19. The agent framework used throughout the pipeline is OpenHands.

A.4 Generation Models, Versions, and Yield

DeepSeek-V3.2 (Liu et al., 2025) is used for inversion-prompt generation, environment inversion, repair-issue generation, and repair rollout. Direct language-model calls for inversion prompts and repair issues use temperature 1.0. Environment inversion and repair rollout run through OpenHands 1.1.0 with greedy decoding and temperature 0. The execution harness uses Terminal-Bench 0.2.0.

To test whether the inversion agent can be replaced, we run two agents on the same 512 pre-generated inversion prompts. Table 7 reports successful environment construction, which measures yield rather than overall task quality.

Table 7: Environment-inversion yield on 512 matched prompts.

Inversion agent	Successful environments	Yield
Qwen3-Coder-30B-A3B-Instruct	63/512	12.3%
DeepSeek-V3.2	131/512	25.6%

A.5 Expert Audit Protocol and Results

Three non-author experts independently evaluate binary criteria for *realism*, *solvability*, *issue–test consistency*, and *shortcut resistance*. Borderline cases default to fail. The audit is stage-blind and partially source-blind: raters are not told the CLI-Gym funnel stage, receive the same rubric across datasets, and do not receive the paper's claims or results, but we do not claim full source anonymity. Rates are means over expert–example judgments.

For each CLI-Gym stage, the target sampling fraction is approximately 10%. Samples are stratified across the 29 repositories: integer quotas are allocated proportionally with largest-remainder (Hamilton) rounding, followed by random sampling without replacement within each repository using seed `20260726`. Fleiss' κ, computed over all 1,160 example–dimension units, is approximately 0.72 ($p < 0.001$). The anonymous artifact provides sampling manifests, item-level labels, example-clustered bootstrap 95% confidence intervals, and scripts that reproduce every statistic.

Table 8: Expert-audit point estimates. External datasets are descriptive references with different population sizes and sampling fractions; the table does not claim statistical equivalence or uniform superiority.

Stage or dataset	Population	Audited	Sampling rate	Realism	Solvability	Consistency	Shortcut resistance
CLI-Gym generated tasks	1,655	160	9.7%	70.0%	59.8%	80.0%	79.8%
CLI-Gym rollout-solved tasks	417	40	9.6%	81.7%	98.3%	96.7%	78.3%
CLI-Gym filtered tasks	291	30	10.3%	84.4%	97.8%	95.6%	88.9%
TerminalTraj	20,000	30	0.15%	65.6%	71.1%	82.2%	84.4%
Terminal-Bench 1.0	80	30	37.5%	92.2%	97.8%	97.8%	97.8%

The audit begins with the 1,655 assembled issue–environment–verifier tuples. Unsuccessful inversion attempts are not pooled into the quality rates because they do not form complete task instances. Common unsuccessful-generation modes include inducing no test failure, producing a Dockerfile that cannot replay the interactive state, preventing the test runner from starting, generating an issue inconsistent with the verifier, leaving a shortcut, and constructing an implausible sabotage scenario. These modes motivate both automatic filtering and the separate expert audit.

A.6 Resource Accounting

The 4,066 generation attempts consume approximately 2.3 billion language-model tokens, and each inversion episode has an approximately 15-minute time limit. Training Qwen3-32B uses 16 NVIDIA A800 GPUs for approximately two hours, while training Qwen3-235B-A22B-Instruct uses 64 A800 GPUs for approximately four hours. Repository selection, system implementation, expert auditing, and artifact preparation also require human labor, but complete retrospective person-hour accounting is unavailable. We therefore report these dimensions separately and do not compare tokens directly with contributor counts or claim a human-free pipeline.

B Detailed Experiments

In this section, we present the full experimental configurations and execution details.

B.1 Training Set Construction

Starting from 29 gold instances, we generate 4,066 inversion prompts, which result in 1,655 verified problem instances containing a replayable image, issue, and verifier. DeepSeek-V3.2 with OpenHands solves 417 tasks. We filter 126 trajectories and retain 291 Filtered-Success trajectories for training.

The filtering criteria are as follows. (1) Trajectories with fewer than 20 steps. We consider the number of interaction steps to be correlated with task difficulty; therefore, trajectories that solved tasks with few steps were removed, as they typically correspond to trivial or low-difficulty environments. (2) Cheating trajectories. We discarded trajectories that exploited historical artifacts such as cached Git information, Conda logs, or other unintended shortcuts to solve the task, bypassing the intended problem-solving process.

B.2 Training Details

For Qwen3-32B, the learning rate is initialized at 2×10^{-5} and follows a cosine decay schedule. A linear warmup covers the first 5% of training steps, increasing the learning rate from 1×10^{-6} to 2×10^{-5}. We train for 10, 15, and 20 epochs with batch size 16. We use the Qwen3-Coder agent template and a maximum sequence length of 100k tokens, extending the native 40k context window with YaRN at factor 2.5.

For Qwen3-235B-A22B-Instruct, the learning rate is initialized at 1×10^{-5} with the same cosine schedule and 5% linear warmup from 1×10^{-6}. We train for 10, 15, and 20 epochs, use the Qwen3-Coder agent template, and set the maximum sequence length to 100k tokens.

For training, we use 16/64 NVIDIA A800 GPUs for approximately 2/4 hours for Qwen3-32B/Qwen3-235B-A22B-Instruct, respectively.

Checkpoint selection. For candidate epoch e, let $L_{e,\mathrm{start}}$ and $L_{e,\mathrm{end}}$ be the training losses at the beginning and end of the epoch. We compute

$$r_e = \frac{|L_{e,\mathrm{end}} - L_{e,\mathrm{start}}|}{\max\left(|L_{e,\mathrm{start}}|, \epsilon\right)}. \tag{5}$$

Only checkpoints satisfying $r_e < 0.01$ enter the candidate set. Among qualifying checkpoints, we select the checkpoint with the smallest r_e; exact ties are resolved in favor of the earlier epoch. This deterministic loss-based rule does not use Terminal-Bench scores or qualitative inference behavior.

B.3 Agent and Inference Details

We adopt the OpenHands agent framework Wang et al. (2025) as the execution interface between language models and CLI environments. OpenHands provides a unified Agent–Computer Interface (ACI) that enables models to interact with containerized systems through structured tool calls, including shell command execution, file editing, and environment inspection. At inference time, each model operates as a single autonomous agent that iteratively observes environment feedback and issues actions until the task is solved or a termination condition is reached.

Action space. The agent is allowed to invoke the tools in Figure 20, All actions are executed inside isolated Docker containers constructed for each task.

Termination conditions. An episode terminates when one of the following conditions is met: (1) the agent explicitly calls the finish tool; (2) a global time limit setting by Terminal-Bench is exceeded.

Decoding and aggregation. During inference, we use greedy decoding with temperature 0 and a maximum context length of 128k tokens. Pass@1 is one standalone run per task. Pass@3 is a separate experiment with three new runs per task and counts success if any run passes the verifier.

B.4 Train–Evaluation Overlap Analysis

We compare CLI-Gym against Terminal-Bench 1.0 and 2.0 jointly at five levels. Table 9 finds one shared repository name and no matches at the other checked levels. The repository-name match uses different commits, images, tests, and tasks.

Table 9: Checked overlap between CLI-Gym and Terminal-Bench 1.0/2.0.

Comparison level	Matches
Repository name	1
Commit SHA	0
Docker image digest	0
Base image	0
Unit-test path or test content	0

These checks find no direct instance-level overlap at the measured levels. They cannot establish that the base models never encountered related public repositories during pretraining.

B.5 Ablation Studies and Case Visualization

Unless otherwise stated, every ablation uses the checkpoint-selection rule in Appendix B.2: among the 10-, 15-, and 20-epoch candidates satisfying $r_e < 0.01$, we select the checkpoint with the smallest r_e, breaking exact ties in favor of the earlier epoch. No Terminal-Bench score or qualitative inference behavior is used for checkpoint selection.

Agentic Coding Pretraining with SWE Tasks. The open-source SWE-style trajectories initialize general agentic coding abilities such as repository navigation, tool invocation, and multi-step program repair. The release artifacts document their sources, licenses, and preprocessing. Direct train–evaluation overlap checks for CLI-Gym are reported in Table 9; as with any public-repository evaluation, these checks cannot rule out pretraining exposure. In this setting, models are first trained on the SWE-style trajectories for one epoch, followed by fine-tuning on the Filtered-Success trajectories. The second-stage checkpoint is selected using the same loss-stability rule as the main experiments. All other training hyperparameters, including learning rate, batch size, context length, optimizer, and agent interface, are kept identical to the main setting.

Environment Diversity via Scaling Repositories. When multiple repositories are involved, trajectories are uniformly sampled to ensure balanced coverage across repositories, avoiding dominance by any single codebase or environment configuration. For every repository subset, the checkpoint is selected using the same loss-stability rule, with all other settings held fixed.

Filtering Low-Quality Trajectories. For both the Filtered-Success and Raw-Success datasets, the checkpoint is selected solely using the same within-epoch loss-stability criterion. All other factors are kept identical; Terminal-Bench performance is not used to select an epoch.

Category-wise Improvement. We compare Qwen3-32B and LiberCoder-32B on Terminal-Bench 1.0 using the categories in the official registry. We report pass@3 for this descriptive analysis.

Failure Type Distribution. We analyze unresolved tasks from one full Terminal-Bench 1.0 run for Qwen3-235B-A22B-Instruct and LiberCoder-235B-A22B and categorize their observed failure modes.

B.6 Qualitative Behavior Cases

Two cases illustrate changes in diagnostic behavior without establishing a general causal decomposition. On `cron-broken-network`, the base model repeatedly retries requests, attempts to disable a VPN, or asks the user for more information. After CLI-Gym training, the model treats the failure as environmental, checks lower-level network tools and services such as `ping` and `curl`, and localizes the fault more systematically. On `fix-permissions`, the base model debugs the contents of a non-executable script, whereas the trained model first checks ownership and execution permissions. These examples are consistent with improved environment diagnosis, while the quantitative results may also reflect better tool use and long-horizon control.

B.7 More Experiments

Impact of Different Agents

Table 10 gives two leaderboard examples showing that the agent scaffold can materially change the score for a fixed model. These cross-scaffold values are descriptive rather than controlled measurements from our pipeline.

Table 10: Performance comparison on Terminal-Bench 2.0.

Claude Haiku 4.5	
Agent	Score
Terminus 2	28.3
Claude Code	27.5
OpenHands	13.9

Claude Opus 4.5	
Agent	Score
Terminus 2	57.8
Claude Code	52.1
OpenHands	51.9

Comparison with other datasets. Table 11 compares representative datasets and benchmarks in terms of the number of task instances, base environments, and storage footprint.

CLI-Gym derives multiple problem instances from each base environment. Storage values in Table 11 provide resource context, but differences in task scope and packaging prevent a direct quality or cost comparison.

Table 11: Comparison of representative datasets in terms of scale and storage footprint. We report the number of task instances (#Instance), the number of base images/environments (#Images), and the required storage size.

DataSet	#Instances	#Images	Size
Code-centric			
SWE-gym	2438	2438	6 TBs
SWE-smith	50137	128	295 GBs
R2E-gym	8135	–	4 TBs
CLI-centric			
Terminal-Bench 1.0	80	14	192 GBs
Terminal-Bench 2.0	89	11	235 GBs
Ours	1655	29	119 GBs

Gain with Hint. We conducted an ablation study to analyze the effect of introducing hints when generating problem statements in CLI-Gym. Specifically, this variant augments the repair issue with additional hints extracted from the task prompt that induces failure, with the goal of facilitating trajectory generation and increasing data yield.

As shown in Table 12, adding hints significantly increases the number of usable trajectories, enabling us to collect 291 trajectories instead of 104, which leads to a substantial performance improvement. When controlling for data scale by subsampling the full dataset to the same size (104 trajectories), the performance remains comparable.

Table 12: Ablation study on hint-augmented repair issue generation. Adding hints substantially increases the number of valid trajectories produced by CLI-Gym, leading to improved downstream performance. When controlling for data scale, hints alone do not significantly affect performance.

Setting	# Trajectories	Performance
w/o hints	104	23.0
w/ hints (subsampled)	104	22.8
w/ hints (full)	291	32.4

Interaction Efficiency. Figure 9 reports the distribution of interaction steps for solved tasks on Terminal-Bench 1.0, comparing our model with Claude Sonnet 4.5.

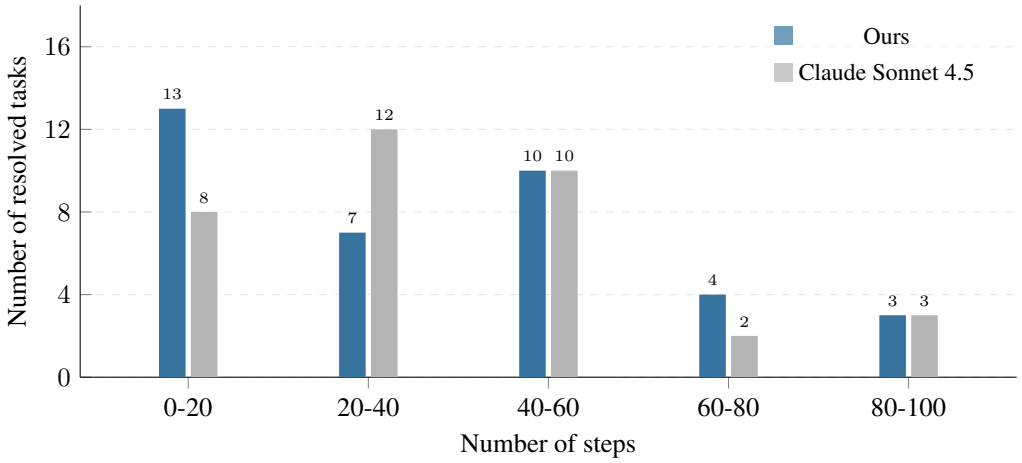


Figure 9: Distribution of interaction steps for solved tasks on Terminal-Bench 1.0.

Performance Benefits Model Behavior. Figure 10 analyzes the relationship between overall performance and the frequency of stuck failures. We observe a strong negative correlation between pass@1 performance and the proportion of trajectories in which the agent becomes stuck in repetitive action loops. As the number of Filtered-Success trajectories increases, model performance steadily improves, while the incidence of looped behavior drops sharply from 42.7% to 3.0%. This result suggests that environment-repair supervision not only improves success rates, but also substantially enhances the agent's ability to escape unproductive interaction patterns and maintain effective long-horizon control.

C LIMITATIONS

Several limitations bound the conclusions of this study.

- **Teacher-selection bias.** SFT uses only tasks solved by the DeepSeek-V3.2 rollout policy and retained by trajectory filtering. The 291 trajectories therefore overrepresent tasks within the teacher's capability distribution.
- **Generator dependence.** The matched-prompt experiment shows that inversion yield varies substantially by generator. Task types and failure modes may likewise reflect DeepSeek-V3.2's capabilities and preferences.
- **Task-quality heterogeneity.** The full collection has lower audited solvability and realism than the rollout-solved and filtered subsets. A stratified sample characterizes population-level quality but does not certify every task.
- **Construct impurity.** Only 48.1% of tasks are environment-only; 19.0% are source-only and 32.9% are mixed. Results should not be attributed exclusively to pure environment diagnosis.

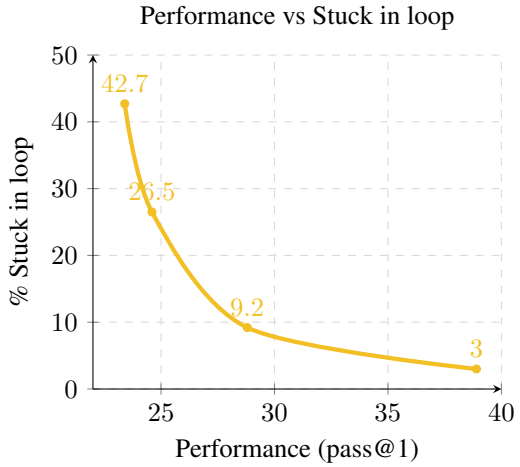


Figure 10: Relationship between task performance and the proportion of failure cases stuck in action loops. Each point corresponds to a Qwen3-32B model fine-tuned with an increasing number of filtered-success trajectories and evaluated on Terminal-Bench 1.0. The x-axis reports pass@1 performance, and the y-axis shows the percentage of tasks where the agent becomes stuck in repetitive loops.

Table 13: Statistics and descriptions of 29 repositories we used (total task instances: 1655).

Repo	Unit Tests	Task Instances	Description
apispec	569	102	Pluggable API specification generator for web frameworks.
autograd	585	20	Efficiently computes derivatives of numpy code automatically.
cantools	372	123	Tools for CAN bus signal decoding, encoding and editing.
click	499	2	Composable command line interface toolkit for Python.
cookiecutter	356	4	Utility for creating projects from project templates.
dominate	65	41	Python library for creating and manipulating HTML documents.
environs	105	2	Simplified environment variable parsing for Python apps.
exceptiongroup	89	89	Backport of PEP 654 exception groups for older versions.
faker	2103	73	Generates fake data for testing, databases, and filling.
feedparser	4229	4	Robust RSS and Atom feed parser for Python developers.
flashtext	39	11	Efficient keyword search and replacement in large texts.
fvcore	140	59	Shared toolkit used across Facebook AI Research projects.
glom	178	23	Powerful tool for restructuring and transforming data.
godotenv	52	3	Loads environment variables from .env files into system.
gspread	149	27	Python wrapper for Google Sheets API v4 interaction.
gunicorn	248	164	Pre-fork worker model WSGI HTTP Server for UNIX.
h11	76	27	Pure-Python, zero-I/O implementation of HTTP/1.1.
iniconfig	28	2	Small and simple configuration parser for .ini files.
markupsafe	70	59	Safely escapes characters for use in HTML and XML.
modin	409	5	Scales pandas workflows by changing a single line of code.
pandas	109403	248	High-performance data analysis and manipulation library.
ptyprocess	12	112	Launches a subprocess in a pseudo terminal (pty).
pyopenssl	449	35	Python interface to the OpenSSL cryptography library.
python-dotenv	178	73	Reads key-value pairs from .env files for local dev.
routers	23	25	Lightweight and fast routing implementation for web.
scrapy	2882	122	High-level web crawling and data scraping framework.
slugify	81	3	Converts strings to clean, URL-friendly slug strings.
textdistance	427	84	Computes distance between sequences with many algorithms.
tweepy	148	113	Python library for interacting with the Twitter API.

Degradation Issue Generation Prompt 1

```
# Role: Chaos Engineering Lead and QA Expert
# Background: We are testing the OpenHands agent in a secure Docker container.
# Objective: Design a one-way or multi-directional environmental disruption task based
     on the Disruption Direction, and select some unit tests from the List of Candidate
      UTs to verify whether the disruption was successful.

## Input Data
List of Candidate UTs:
{candidate_uts_list}

## Disruption Direction
{directions}

## Constraints (Key)
1. **Natural Language**: The task description must be a clear, human-readable goal (e.g
     ., "perform certain actions to achieve a specific objective"). Some bash commands
     for reference can be provided to help the agent complete the disruption task.
2. **Causality**: The chosen disruptions must logically cause the selected UT (unit
     test) to fail, and no more than 50 UTs should be selected.
3. **Complexity**: The generated disruption tasks should have a certain level of
     difficulty to solve. They should also not leave backup files or allow bypassing
     expected recovery methods.
In addition, they should involve recovery challenges such as:
    - Tampering with system paths/files, causing kernel/system issues, e.g., VFS:
         unable to mount root filesystem, with the error message "unknown-block(0, 0)".
          Do not simply mimic this issue.
    - Encrypting documents that are difficult to obtain through other means.
    - ...{more examples}
    (Do not limit yourself to the above examples.)
5. **Diversity**: I have already generated the following tasks, please do not generate
     tasks with similar themes. The methods of causing damage do not necessarily have
     to be related to Python, nor do they necessarily need to be implemented using
     Python. Think outside the box.
## Generated Tasks
{os.listdir(dataset_path)}

## Output Format
Strictly follow the following Markdown format:
---
**Task Name**: <Short Title>
**Category**: <Single word, e.g., Data>
**Selected UTs**:
- <Path to UT 1>
- <Path to UT 2>
**Task Description**: <Detailed natural language instructions provided to the agent.
     Describe the **goal** and **steps** to create the vulnerability, and let the agent
      verify the vulnerability.>
**Expected Result**: <The error that should occur>
**Recovery Strategy**: <How to fix it>
---
```

Figure 11: Degradation Issue Generation Prompt 1

Degradation Issue Generation Prompt 2

```
# Role: Chaos Engineering Lead (Refinement)
# Context: Refine the draft sabotage task to be a high-quality
    directive for an AI Agent.

## Input Data
1. **Target UTs (Paths)**:
{extract_result['selected_uts']}

2. **Draft Task**:
{extract_result}

## Refinement Objectives
1. **Natural Language Refinement**: Ensure `Task Description` is
    written in clear, instructional English. It should explain *what
    * to do (the objective) rather than just listing code.
    * Bad: "Run `chmod 000 file`"
    * Good: "Locate the log file and modify its permissions so that
        it is not writable by the current user."
2. **Verification Step**: Add a sentence at the end of `Task
    Description` asking the agent to: "Run the target Unit Tests to
    confirm the failure."
3. **Logic Check**: Ensure the task strictly targets the logic of
    the provided "Target UTs".

## Output Format
Retain the exact Markdown format.

---
**Task Name**: <Refined Title>
**Category**: <Same>
**Selected UTs**:
<List the input Target UTs paths here, one per line>
**Task Description**: <Refined natural language objective +
    verification request>
**Expected Result**: <Detailed error>
**Recovery Strategy**: <Refined recovery>
---
```

Figure 12: Degradation Issue Generation Prompt 2

Degradation Instruction Template

```
instruction: |
  You are in a completely isolated and secure Docker container,
      where you can do anything to complete the following task:
    {degradaton_issue}

  I will run these tests using a conda environment named "testbed"
      with the command "cd /testbed && source /opt/miniconda3/bin/
      activate && conda activate testbed && pytest [UTs]." I will
      confirm whether you have successfully completed the task by
      observing whether any tests fail. Therefore, please use the
      Python environment located in /opt/miniconda3/envs/testbed/bin
       throughout the entire process.
  Available UTs:
    {UTs}

  After completing the task, please summarize a Dockerfile. I want
      to append this Dockerfile to the original image's Dockerfile
      that you had before executing the task, so that I can
      reproduce the environment after the task was executed. Based
      on your memory, write all meaningful operations you performed
      from start to finish in the form of Dockerfile commands in the
       /logs/Dockerfile file. You do not need to include any
      commands you did not execute, nor do you need to construct a
      fully functional Dockerfile from scratch because I will append
       it to the original image. Also, there is no need to guess how
       the original image was built; just write down the commands
      you actually executed. Since my entire process is automated,
      no comments or explanations are required. Pay attention to
      writing multi-line instructions in accordance with Dockerfile
      syntax.

  Do not leave any introductory documents or code comments (e.g.,
      INTRODUCTION.md)! Do not leave backup files (e.g., init.py.bak
       and file in /tmp). Do not make the environment look like it
      has been deliberately sabotaged (e.g., by adding a
      sitecustomize.py); make the sabotage appear more natural.
      After tampering with the files, do not change their names, as
      this will alert the forensics experts. Remember to delete
      files like .git, conda package cache and so on, as they can
      make solving the problem too easy.
```

Figure 13: Degradation Instruction Template

Agent Working Demo


```
{
    "id": 69,
    "source": "agent",
    "message": "Command `cat > /etc/profile.d/02-set-interval-locale.sh << 'EOF'
         nexport LC_ALL=en_US.interval nexport LANG=en_US.interval nexport LC_NUMERIC=
         en_US.interval nexport LC_MONETARY=en_US.interval EOF` executed with exit code
          0.",
    "action": "run",
    "function": {
        "arguments": "{"command": "cat > /etc/profile.d/02-set-interval-locale.sh << '
             EOF'  nexport LC_ALL=en_US.interval nexport LANG=en_US.interval nexport
             LC_NUMERIC=en_US.interval nexport LC_MONETARY=en_US.interval EOF", "
             security_risk": "MEDIUM"}",
        "name": "execute_bash"
    },
    ...
}
...
{
    "id": 131,
    "source": "agent",
    "message": "All done! What's next on the agenda?",
    "action": "finish",
    "function": {
        "arguments": "{"message": "Task completed successfully. I have introduced
             subtle numeric parsing corruption through locale manipulation that will
             cause pandas IntervalIndex operations to fail when comparing and
             constructing interval boundaries.**What was done:**1. **Created corrupted
             locale `en_US.interval`**: Generated a custom locale variant with swapped
             decimal point (comma) and thousands separator (period), conflicting with
             Python's default numeric parsing expectations.2. **Modified LC_NUMERIC and
              LC_MONETARY files**: Set decimal_point to "," and thousands_sep to "." in
              both numeric and monetary formatting rules.3. **Disabled system locale
             archive**: Renamed `/usr/lib/locale/locale-archive` to force dynamic
             loading of individual locale directories.4. **Set system-wide environment
             variables**: Created `/etc/profile.d/02-set-interval-locale.sh` to export
             LC_ALL, LANG, LC_NUMERIC, and LC_MONETARY to use the corrupted locale.5.
             **Poisoned Python's numeric parsing cache**: Created `sitecustomize.py` in
              the testbed conda environment's site-packages that:   - Sets the
             corrupted locale on Python interpreter startup   - Calls C library's `
             strtod` function with ambiguous numeric strings like "1,5", "2.3",
             "3,14159" to create cached parsing states6. **Cleaned up forensic traces
             **: Removed pip cache, conda cache, conda package cache, and the .git
             directory.. The environment is now ready. You can run the pandas
             IntervalIndex unit tests using the testbed conda environment to observe
             the numeric parsing corruption causing interval boundary comparison and
             construction errors."}",
        "name": "finish"
    },
    ...
}
```


Figure 14: Agent Working Demo

Repair Issue Generation Prompt 1

```
# Role: QA Engineer / Bug Reporter
# Objective: Report a bug to the AI agent without revealing the
    solution, forcing the agent to investigate.

## Background (Input Data)
**Malicious Behavior (Root Cause: The current environment is a test
    environment that has been affected by the malicious task
    described as follows):**
{data["task_description"]}
**Unit Tests that actually failed (should be mentioned in the bug
    report):**
{symptoms_UTs}

## Your Task
Write a natural language bug report to the agent.
1. **Describe the Symptoms**: Summarize a recoverable issue.
    Describe what went wrong. Point out that some unit tests failed.
2. **Objective**: Ask the agent to **investigate the environment**,
    identify the root cause, and fix it.
3. **Constraints**: Do not provide a solution or analyze the problem
    ; just describe what happened.

## Output Rules (Strict)
- **Output only the instruction text.**
- Do not use Markdown headings, do not use "Task Name," and do not
    use code blocks.
- The output should look like a user asking for help: "I can't...
    Can you figure out the reason and help me fix this bug?"

## Output Format (Strict)
<Help information, only mentioning the failed unit tests>
Hint:
<Describe where the problem might be based on the description of the
     malicious task, but do not provide a solution. Do not specify
    exactly where or what the problem is!>
```

Figure 15: Repair Issue Generation Prompt 1

Repair Issue Generation Prompt 2

```
# Role: QA Engineer / Bug Reporter
# Objective: Report issues to the AI agent without revealing the
    solution, forcing the agent to investigate.

## Background (Input Data)
**Malicious Behavior (Root Cause: The current environment is a test
    environment that has been affected by the malicious task
    described as follows):**
{data["task_description"]}
**Unit Tests that actually failed (should be mentioned in the bug
    report):**
{symptoms_UTs}

## Your Task
Write a natural language help message for the agent.
1. **Direct Request for Help:** State that I have some UTs that are
    failing and need help to fix them.
2. **Describe the Symptoms:** Summarize a recoverable problem.
    Describe what went wrong and what kind of help is needed.
3. **Constraints:** Do not provide a solution or analyze the problem
    . Just describe what happened.

## Output Rules (Strict)
- **Output only the instruction text.**
- Do not use Markdown headings, do not use "Task Name," and do not
    use code blocks.
- I will replace it into a YAML file. Pay attention to indentation
    and symbol conventions (e.g., do not use "-").
- The output should look like a user asking for help: "I can't...
    Can you help me figure out why and tell me how to fix it?"

## Output Format (Strict)
<Help message, only referring to the UTs that are failing>
hint:
<Describe where the problem might be based on the description of the
     malicious task, but do not provide a fix. Do not specify
    exactly where or what the problem is!>
```

Figure 16: Repair Issue Generation Prompt 3

Repair Issue Generation Prompt 3

```
# Role: QA Engineer / Bug Reporter
# Objective: Report issues to the AI agent without revealing the
    solution, forcing the agent to investigate.

## Background (Input Data)
**Malicious Behavior (Root Cause: The current environment is a test
    environment that has been affected by the malicious task
    described as follows):**
{data["task_description"]}
**Unit Tests that actually failed (should be mentioned in the bug
    report):**
{symptoms_UTs}

## Your Task
Write a natural language help message for the agent.
1. **Direct Request for Help:** State that I have some UTs that are
    failing and need help to fix them.
2. **Describe the Symptoms:** Summarize a recoverable problem.
    Describe what went wrong and what kind of help is needed.
3. **Constraints:** Do not provide a solution or analyze the problem
    . Just describe what happened.

## Output Rules (Strict)
- **Output only the instruction text.**
- Do not use Markdown headings, do not use "Task Name," and do not
    use code blocks.
- The output should look like a user asking for help: "I can't...
    Can you help me figure out why and tell me how to fix it?"

## Output Format (Strict)
<Help message, only referring to the UTs that are failing>
hint:
<Describe where the problem might be based on the description of the
     malicious task, but do not provide a fix. Do not specify
    exactly where or what the problem is!>
```

Figure 17: Repair Issue Generation Prompt 3

Repair Instruction Template

```
instruction: |
{problem_statement}
```

Figure 18: Repair Instruction Template

Run-tests Template

```
set -uo pipefail -x

cat > tester.py <<EOF
import os
import subprocess, sys
sys.path.insert(0, '/testbed')

def run_and_log(cmd, log_path):
    with open(log_path, "w", buffering=1, encoding="utf-8") as logf:
        p = subprocess.Popen(
            cmd,
            stdout=subprocess.PIPE,
            stderr=subprocess.STDOUT,
            text=True,
            bufsize=1,
            shell=True,
            executable="/bin/bash"
        )
        for line in p.stdout:
            line = line.replace("\r", "\n")
            sys.stdout.write(line)
            logf.write(line)
        return p.wait()

run_and_log(
    'source /opt/miniconda3/bin/activate; conda activate testbed;
        pytest --disable-warnings --color=no --tb=no --verbose {UTs
        }',
    "/test.log"
)
EOF

chmod +x tester.py
python tester.py
```

Figure 19: Run-tests Template

Tools

```
"function": {
    "name": "execute_bash",
    "description": "Execute a bash command in the terminal within a persistent shell session.###
          Command Execution* One command at a time: You can only execute one bash command at a time.
          If you need to run multiple commands sequentially, use `&&` or `;` to chain them together.*
          Persistent session: Commands execute in a persistent shell session where environment
          variables, virtual environments, and working directory persist between commands...",
},
"function": {
    "name": "think",
    "description": "Use the tool to think about something. It will not obtain new information or make
          any changes to the repository, but just log the thought. Use it when complex reasoning or
          brainstorming is needed.Common use cases:1. When exploring a repository and discovering the
          source of a bug, call this tool to brainstorm several unique ways of fixing the bug, and
          assess which change(s) are likely to be simplest and most effective.2...",
},
"function": {
    "name": "finish",
    "description": "Signals the completion of the current task or conversation.Use this tool when:-
          You have successfully completed the user's requested task- You cannot proceed further due to
           technical limitations or missing information.The message should include:- A clear summary
          of actions taken and their results- Any next steps for the user- Explanation if you're
          unable to complete the task- Any follow-up questions if more information is needed",
},
"function": {
    "name": "execute_ipython_cell",
    "description": "Run a cell of Python code in an IPython environment.* The assistant should define
          variables and import packages before using them.* The variable defined in the IPython
          environment will not be available outside the IPython environment (e.g., in terminal).",
},
"function": {
    "name": "task_tracker",
    "description": "This tool provides structured task management capabilities for development
          workflows.It enables systematic tracking of work items, progress monitoring, and efficient
          organization of complex development activities.The tool maintains visibility into project
          status and helps communicate progress effectively to users.## Application Guidelines Utilize
           this tool in the following situations:...",
},
"function": {
    "name": "str_replace_editor",
    "description": "Custom editing tool for viewing, creating and editing files in plain-text format*
          State is persistent across command calls and discussions with the user* If `path` is a text
          file, `view` displays the result of applying `cat -n`. If `path` is a directory, `view`
          lists non-hidden files and directories up to 2 levels deep* The following binary file
          extensions can be viewed in Markdown format: [".xlsx", ".pptx", ".wav", ".mp3", ".m4a", ".
          flac", ".pdf", ".docx"]. IT DOES NOT HANDLE IMAGES.* The `create` command cannot be used if
          the specified `path` already exists as a file* If a `command` generates a long output, ...",
}
```

Figure 20: Tools

Dockerfile Demo

```
FROM task-pandas:latest

RUN mkdir -p /corrupted_libs
RUN cp /opt/miniconda3/envs/testbed/lib/libsqlite3.so.0.8.6 /
    corrupted_libs/libsqlite3.so.0
RUN cp /opt/miniconda3/envs/testbed/lib/libz.so.1.2.13 /
    corrupted_libs/libz.so.1
RUN dd if=/dev/zero of=/corrupted_libs/libsqlite3.so.0 bs=1 count=24
     seek=8 conv=notrunc
RUN dd if=/dev/zero of=/corrupted_libs/libz.so.1 bs=1 count=24 seek
    =8 conv=notrunc
RUN rm /opt/miniconda3/envs/testbed/lib/libsqlite3.so.0.8.6
RUN cp /corrupted_libs/libsqlite3.so.0 /opt/miniconda3/envs/testbed/
    lib/libsqlite3.so.0.8.6
RUN rm /opt/miniconda3/envs/testbed/lib/libz.so.1.2.13
RUN cp /corrupted_libs/libz.so.1 /opt/miniconda3/envs/testbed/lib/
    libz.so.1.2.13
```

Figure 21: Dockerfile Demo

- **Test-runner semantics.** A runner failure is an accepted environment-failure outcome because it demonstrates damage to the runtime or test infrastructure. However, it provides no per-test functional outcome: the selected tests remain verification targets but are not labeled as fail-to-pass or pass-to-pass.
- **Model and scaffold coverage.** Experiments cover two Qwen3 scales and one OpenHands scaffold. They do not establish transfer to other model families, tool interfaces, or agent scaffolds.
- **Checkpoint criterion.** Checkpoints are chosen by the reported within-epoch loss-stabilization rule. This objective rule is reproducible but may not select the checkpoint that transfers best to every downstream setting.
- **Pretraining exposure.** The overlap analysis finds no direct match in commits, images, or tests, but cannot show that base models never encountered related public repositories during pretraining.
- **Domain coverage.** The 29 manually curated Python repositories do not represent the full range of CLI work, including embedded toolchains, interactive applications, and non-Python ecosystems.
- **Context efficiency.** Trained models sometimes explore for longer and exceed the 128k inference context more often, motivating work on more efficient search and stopping behavior.

D Impact Statement

CLI-Gym aims to make CLI-agent research more reproducible by releasing a task-construction pipeline, executable environments, and LiberCoder checkpoints. This can lower the barrier to studying software-maintenance and system-administration agents. The same capabilities and failure-induction prompts could be misused for intrusive operations or sabotage. All experiments run in isolated Docker containers, and released artifacts should include dual-use documentation and encourage least-privilege sandboxing, monitoring, and human approval for consequential actions. Repository licenses and code provenance must be preserved in downstream use.