ORIGENT: POST-TRAINING HARDWARE ENGINEERING AGENTS THROUGH SCALABLE ENVIRONMENT SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM-based coding agents have made substantial progress in solving software engineering tasks. Applying these agents to hardware engineering requires them to reason about circuit behavior, use simulation and electronic design automation (EDA) tools, and coordinate changes across register-transfer-level (RTL) designs, configuration files, and verification code. Learning these workflows requires hardware-specific interaction experience, but much of the relevant development data is proprietary, limiting public training resources. We introduce OriGent, a framework that constructs executable training environments from the commit history of public hardware repositories. Starting from merged pull requests, OriGent reconstructs repository states, builds target tests validated to fail before the reference repair and pass afterward, and generates problem statements aligned with the tested behavior. Separating reusable environment setup from task-specific construction yields over 18,000 executable tasks from 90 repositories spanning RTL design, verification, EDA software, hardware domain-specific languages, and firmware. We post-train Qwen3.6-35B-A3B through supervised fine-tuning on successful teacher trajectories collected in these environments, followed by online reinforcement learning. On HWE-Bench, a repository-level hardware engineering benchmark, the resolve rate increases from 39.3% for the base model to 55.9% after supervised fine-tuning and 64.3% after reinforcement learning. OriGent outperforms KAT-Coder-V2.5-Dev (51.3%), which is post-trained for agentic coding from the same base model. On a 200-task SWE-bench Pro subset, OriGent improves resolve rate from 44.0% to 58.0%, showing transfer from hardware-focused training to software engineering.

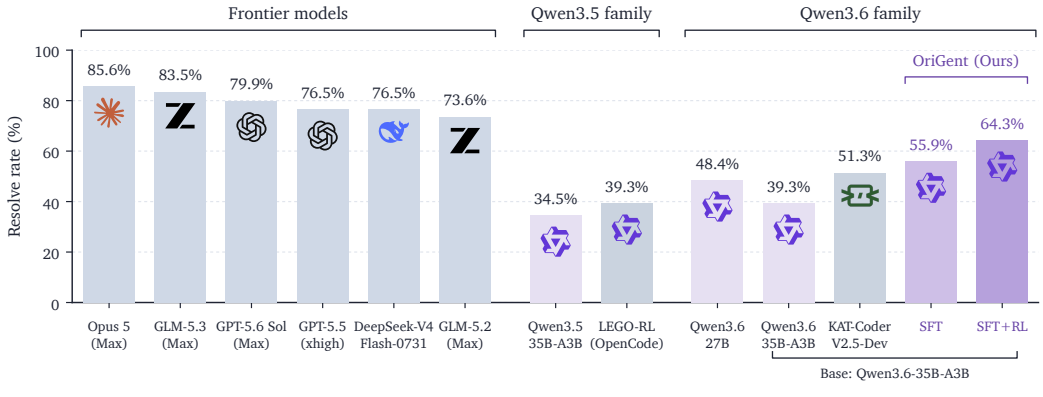


Figure 1: **HWE-Bench performance.** Overall resolve rates on 417 tasks.

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.

1 INTRODUCTION

LLM-based coding agents have become a practical part of software development, resolving reported issues and implementing changes in real-world repositories (Pan et al., 2025; Yang et al., 2025). Hardware engineering poses a different challenge: in complete projects, a single change can span register-transfer-level (RTL) code that describes circuits, configuration files, and verification programs, all built and checked through electronic design automation (EDA) tools. A controller, for example, may compile successfully yet fail a simulation because it transfers data on the wrong clock cycle. Implementing and validating changes in these projects requires reasoning about circuit behavior and design constraints.

Learning these capabilities requires experience with how tool choices and code changes affect build and verification results. Interactive environments have provided such experience in domains like biomedical coding, scientific tool use, and SQL debugging (Xu et al., 2026; Shen et al., 2026; Li et al., 2025a). In hardware, domain-specific supervision has improved Verilog generation and understanding (Cui et al., 2024; Liu et al., 2025c), while execution feedback has supported verification-program generation and revision (Zhang et al., 2026a). These advances motivate synthesizing executable training environments for repository-level hardware engineering, where agents learn through iterative code changes and verification.

Constructing these executable environments at scale presents two challenges. First, public hardware development data are scarce (Liu et al., 2025c). For example, Verilog and SystemVerilog together account for approximately 0.10% of files in the training subset of The Stack v3 (Lozhkov et al., 2026). Second, constructing executable hardware tasks requires establishing reproducible execution conditions and tests that reliably check the intended behavior. Simulator and HDL-library versions, testbenches, and build configurations vary across hardware repositories. Problem statements and behavioral tests need to express the same requirements, so that incorrect repairs are rejected and alternative implementations satisfying those requirements are accepted. This motivates our question: *can interaction experience from synthesized environments improve coding agents' ability to solve repository-level hardware tasks?*

We introduce OriGent, a framework for post-training hardware engineering agents through scalable environment synthesis. Although public hardware development data are limited, merged pull requests provide code changes, discussions, and repository context from which executable tasks can be reconstructed. Construction agents reconstruct historical repository states, build target tests, verify that they fail before the reference repair and pass afterward, and generate problem statements aligned with the tested behavior. To scale construction, OriGent separates reusable toolchain preparation from task-specific setup and verification. The resulting collection contains 18,257 executable tasks from 90 repositories spanning RTL design, verification, EDA software, hardware domain-specific languages, and firmware.

These environments support both the collection of successful teacher trajectories for supervised fine-tuning (SFT) and online interaction with test-based rewards for reinforcement learning (RL). We train OriGent from Qwen3.6-35B-A3B (Qwen Team, 2026b) through these two stages.

On HWE-Bench (Cui et al., 2026), OriGent raises the resolve rate from 39.3% for the base model to 55.9% after SFT and 64.3% after RL (Figure 1). It outperforms KAT-Coder-V2.5-Dev (51.3%), which is post-trained for agentic coding from the same base model (KwaiKAT Team, 2026). The gains extend to software repository repair: on a 200-task SWE-bench Pro subset, OriGent reaches 58.0%, compared with 44.0% for the base model and 48.5% for KAT-Coder-V2.5-Dev.

Our main contributions are as follows:

- A scalable pipeline for synthesizing executable hardware tasks from merged pull requests, separating reusable toolchain preparation from task-specific construction and aligning problem statements with execution-validated tests.
- A hardware training resource for SFT and online RL, comprising 18,257 executable tasks from 90 repositories and 8,013 successful teacher trajectories.
- OriGent, post-trained from Qwen3.6-35B-A3B using this resource. It outperforms its base model on hardware repository tasks, with gains extending to software repository tasks.

2 Related Work

Hardware engineering models and benchmarks. Hardware post-training includes synthetic supervision for RTL generation (Liu et al., 2025b;a) and joint modeling of Verilog understanding and generation (Liu et al., 2025c). Execution feedback supports reinforcement learning for RTL generation (Zhu et al., 2025) and joint training of RTL and reference-model agents in ChipMATE (Yu et al., 2026a). For verification, PRO-V-R1 and CovR train generators using functional and coverage feedback, respectively (Zhao et al., 2026b; Abdelatty et al., 2026). LLM4Cov uses simulation and coverage feedback to supervise testbench revisions within hardware repositories (Zhang et al., 2026a). Benchmarks assess specification-to-RTL generation (Liu et al., 2023; Lu et al., 2024), broader design and verification tasks (Pinckney et al., 2025; Yu et al., 2026b), and formally checked RTL reasoning and repository issues (Wang et al., 2026a). HWE-Bench evaluates historical bug repairs through project-native verification flows (Cui et al., 2026). Our focus is training agents to coordinate code changes and tool use across complete hardware projects.

Executable training environments. SWE-Gym recovers executable environments for historical software issues (Pan et al., 2025); R2E-Gym and Scale-SWE generate tests and problem statements from commits and pull requests, respectively (Jain et al., 2025; Zhao et al., 2026a). Other pipelines construct tasks from repository contents: SWE-smith introduces defects checked by existing tests (Yang et al., 2025), and CodeMidas derives behavioral specifications and executable tests from implemented functionality (Ye et al., 2026). Repository-level reuse of installation and testing procedures supports multilingual task collection in SWE-rebench V2 (Badertdinov et al., 2026). Domain-specific environments also support biomedical coding, scientific tool use, and SQL debugging in MedAgentGym, SciAgentGym, and SWE-SQL, respectively (Xu et al., 2026; Shen et al., 2026; Li et al., 2025a). OriGent reconstructs historical hardware changes under the simulation and build toolchains needed to execute and verify them.

Agent post-training. SWE-Lego studies supervised fine-tuning on trajectories from real and synthetic software tasks, preserving interaction context while masking selected erroneous model responses during training (Tao et al., 2026). SWE-Master continues from supervised training with reinforcement learning on executable repository tasks (Song et al., 2026). LEGO-RL integrates online RL into native coding-agent frameworks (Du et al., 2026), and T1 applies RL to long-horizon terminal interaction (Yang et al., 2026). TMax examines synthetic environment diversity and difficulty alongside transfer to new tasks and agent frameworks (Ivison et al., 2026). For interactions that exceed a context window, CompactionRL and SUPO jointly optimize task execution and summary generation (Li et al., 2026; Lu et al., 2026). OriGent uses its synthesized hardware environments for trajectory SFT and online RL, including supervision of generated summaries.

Table 1 compares representative hardware learning resources and executable training environments.

Table 1: **Comparison of training resources.**

Work	Task scope	Training resource	Training
CircuitNet 3.0 (Wang et al., 2026b)	Timing and power prediction	RTL, netlists, layouts, and labels	Supervised learning
ScaleRTL (Deng et al., 2025)	RTL generation	Specifications, reasoning traces, and RTL	SFT
LLM4Cov (Zhang et al., 2026a)	Testbench generation	Testbench revisions with execution feedback	Multi-stage SFT
R2E-Gym (Jain et al., 2025)	Software repair	Tasks and trajectories from commits	SFT
SWE-smith (Yang et al., 2025)	Software repair	Tasks and trajectories from injected bugs	SFT
CodeMidas (Ye et al., 2026)	Repository coding	Executable tasks from existing functionality	RL
OriGent (Ours)	**Hardware engineering**	**Executable tasks and teacher trajectories**	**SFT + RL**

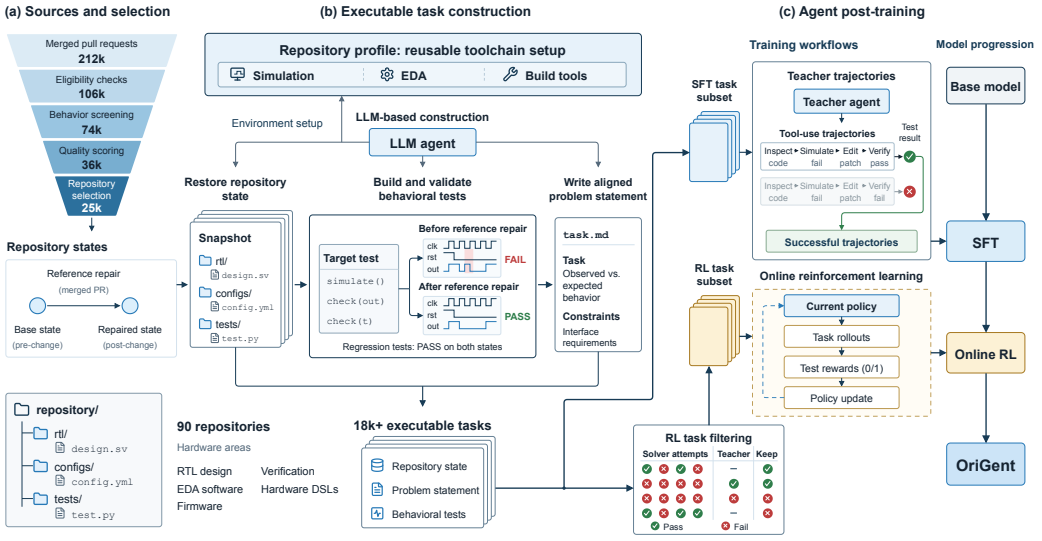


Figure 2: **Overview of OriGent. (a)** Merged pull requests from hardware repositories are screened to select candidate changes, each providing a base state and a reference repair. **(b)** Using reusable toolchain environments, construction agents restore repository states and validate tests before generating problem statements aligned with the tested behavior. **(c)** The model undergoes supervised fine-tuning on successful teacher trajectories, followed by online reinforcement learning on selected tasks with test-based rewards.

3 Scalable Environment Synthesis

The synthesis pipeline converts merged pull requests into executable tasks, each comprising a prepared repository state, a problem statement, and tests for evaluating proposed changes. Toolchain environments are reused across tasks, while repository states and tests are prepared for each change. Problem statements are generated after test validation to align their requirements with the verified behavior. Figure 2 provides an overview.

3.1 Repository and Environment Preparation

Candidate selection focuses on changes whose intended behavior can be identified from development records and checked through execution. The repository pool combines hardware project seeds with GitHub topic and language searches, yielding 150 public repositories from which merged pull requests are collected. The six source repositories of HWE-Bench are excluded during repository discovery to keep training and evaluation repositories disjoint. After filtering by merge date and patch size, an LLM screens out changes dominated by documentation, routine maintenance, or generated code. The remaining candidates are scored from 0 to 2 on task value, behavior clarity, test signal, scope complexity, and interface clarity using the patch, pull request description, and available issue discussions. Candidates must receive 2 for task value and at least 1 on each other dimension, favoring substantive changes with identifiable, testable behavior. Changes requiring physical hardware for validation are excluded. Keeping only repositories with at least 20 qualifying candidates and deferring Zephyr leaves 90 repositories for construction. Each retained pull request supplies a base commit and a reference modification for environment synthesis. Tests need not accompany the pull request, as they are constructed during synthesis (Section 3.2). Figure 2(a) summarizes candidate retention; filtering settings are given in Appendix A.1.

Tasks from the same repository share much of their toolchain. A *profile agent* prepares reusable toolchain environments from the repository's build requirements, with each *repository profile* specifying the tools and dependencies. Profiles can differ across periods of a repository's history: SpinalHDL, for example, uses separate profiles for legacy sbt/JDK 8 and modern sbt/mill/JDK 17 toolchains. The profile agent validates the setup on a representative batch of candidate tasks. For

each candidate change, a separate *task agent* selects an existing profile, restores the base state, and prepares the task-specific build and runtime configuration, reporting toolchain issues that require profile revision. To prevent access to the reference repair through local Git history, we remove the original Git metadata and initialize a fresh repository from the task's base state.

3.2 Test Construction and Problem Statement Generation

In the prepared environment, the task agent uses the reference change and surrounding code to iteratively develop and run *target tests* for the intended behavior and *regression tests* for related functionality that should be preserved. Execution feedback guides revisions to both the tests and task-specific setup. Tests check the required behavior while allowing alternative implementations. Details such as diagnostic text or generated symbol names are checked only when required by the change, its discussion, or a public specification.

Validation requires the target tests to fail on the base state and pass on the repaired state, while the regression tests pass on both. A valid base-state failure comes from an assertion of the intended behavior; build or dependency errors indicate a construction failure. Each of the four checks runs in a fresh container created from the same prepared image, with the reference patch applied only for the repaired-state checks.

After validation, a generation agent drafts the problem statement from the original issue or pull request and the verified behavior. The statement describes the issue and expected behavior, including public interface requirements enforced by the tests. A separate review agent checks the statement against the source material and execution evidence, correcting unsupported claims and ambiguous requirements. Appendix C gives a complete task example.

3.3 Dataset Overview

The collection comprises 18,257 executable tasks from 90 repositories spanning RTL design and IP, simulation and verification, EDA software, hardware DSLs and generators, and firmware and embedded software. These tasks use 152 repository-specific toolchain profiles (Appendix B). EDA software accounts for the largest share of tasks at 36.4%, followed by firmware and embedded software at 32.6% (Figure 3a).

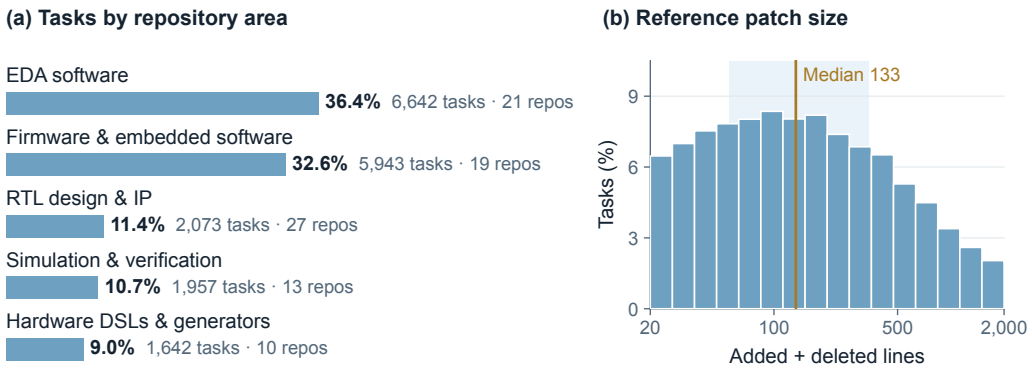


Figure 3: **Dataset overview.** **(a)** Task distribution by repository area. **(b)** Reference patch sizes (added plus deleted lines) on a log scale. Shading indicates the interquartile range.

Reference patches have a median size of 133 lines (interquartile range: 56–345; Figure 3b) and span a median of four files (interquartile range: 2–8).

Changes frequently span multiple files and languages: 89.1% of reference patches modify at least two files, and 36.6% span multiple languages or intermediate representations. The latter proportion reaches 61.8% in EDA software and 56.4% in simulation and verification. These patterns illustrate the breadth of repository-level hardware work represented in the collection, particularly the development of tools used to build and verify hardware designs. Appendix A.2 lists all source repositories, their engineering areas, and their contributions to the SFT and RL training sets.

4 POST-TRAINING HARDWARE ENGINEERING AGENTS

Using the synthesized environments, we post-train Qwen3.6-35B-A3B (Qwen Team, 2026b) through supervised fine-tuning followed by online reinforcement learning, with OpenCode (Anomaly, 2026) as the agent framework in both stages. The two stages use tasks from disjoint sets of repositories.

4.1 SUPERVISED FINE-TUNING

GLM-5.3 (Z.ai, 2026b) generates teacher trajectories by attempting the synthesized tasks. We retain attempts whose final changes pass both target and regression tests, excluding those with more than 300 assistant turns or two or more context compactions (Section 4.2). To avoid exposure to upstream repairs, we also exclude trajectories flagged for external network access.

A turn-level loss mask excludes selected assistant responses in full, including reasoning, text, and tool calls, while retaining the responses and their tool feedback as context for subsequent turns. We apply this mask to incomplete responses, turns lacking reasoning content, and turns with tool-invocation errors identified from the agent framework's error status. Turns that encounter build or test failures remain eligible for supervision, so the model can learn to interpret failures and revise its approach. To limit the verbose reasoning observed in pilot fine-tuning runs, we also mask turns whose reasoning exceeds 16,384 tokens. The final SFT set contains 8,013 successful trajectories from 61 repositories.

4.2 ONLINE REINFORCEMENT LEARNING

Online training continues from the SFT checkpoint with Group Sequence Policy Optimization (GSPO) (Zheng et al., 2025). To select challenging but solvable tasks, we screen candidates with up to four Qwen3.6-27B attempts. We retain tasks with one or two successes in four attempts, or four failures if GLM-5.3 solves the task in the same prepared environment, yielding 2,163 tasks.

For each task, the policy samples a group of trajectories, with binary rewards determined by whether the final changes pass both target and regression tests. Rewards are normalized within each group to obtain trajectory advantages. Groups with identical rewards are filtered out, and additional tasks are sampled to complete the batch (Yu et al., 2025).

Long-running tasks can exhaust the working-context budget before reaching a solution. At the working-context limit, the framework triggers compaction: the policy summarizes its history and resumes from the summary while retaining the repository state (Figure 4). Because subsequent actions depend on the summary, we train summary generation jointly with other responses using the same task reward (Li et al., 2026; Lu et al., 2026). Compaction divides an attempt into training segments. We compute length-normalized GSPO likelihood ratios within each segment, while all segments from the same attempt share its trajectory advantage. Losses are averaged over supervised tokens within each original trajectory, then over trajectories. Appendix D.1 details summary supervision and loss aggregation.

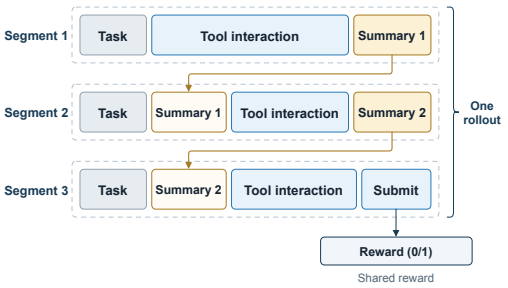


Figure 4: **Context compaction.** The shaded summaries receive policy supervision; outlined copies provide context. All segments share the same reward.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

Benchmarks and models. HWE-Bench (Cui et al., 2026) provides 417 hardware repair tasks from six repositories, all excluded from our training collection. To assess transfer to software repository

tasks, we use the 200-task subset of the original SWE-bench Pro release (Deng et al., 2026) selected by FastContext (Zhang et al., 2026b). We also evaluate on all 89 tasks of Terminal-Bench 2.1 (Merrill et al., 2026) to assess performance on broader terminal tasks.

We compare Qwen3.6-35B-A3B (Qwen Team, 2026b) before post-training, after SFT, and after SFT followed by RL (OriGent). KAT-Coder-V2.5-Dev (KwaiKAT Team, 2026) is an open-weight agentic coding baseline trained through SFT and RL from the same base model. For LEGO-RL (Du et al., 2026), we use the OpenCode checkpoint trained from Qwen3.5-35B-A3B (Qwen Team, 2026a) with GSPO on executable software repository tasks. Both baselines are included in the transfer evaluations alongside the base, SFT, and RL models. HWE-Bench reference models include Claude Opus 5 (Anthropic, 2026), GLM-5.3 (Z.ai, 2026b), GPT-5.6 Sol (OpenAI, 2026b), GPT-5.5 (OpenAI, 2026a), DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026), GLM-5.2 (Z.ai, 2026a), and Qwen3.6-27B (Qwen Team, 2026c).

Training settings. We perform SFT for 3 epochs with a batch size of 32 and a learning rate decaying from 10^{-5} to 10^{-6}. For RL, we use GSPO with a batch size of 32 tasks, 8 rollouts per task, and a constant learning rate of 10^{-6}. We use Miles (Miles Team, 2026) for training, enabling token-in/token-out (TITO) and rollout routing replay (R^3) (Ma et al., 2026) during RL. Full hyperparameters and implementation details are provided in Appendix D.

Table 2: **HWE-Bench results.** Resolve rates (%) by repository and overall.

Model Tasks	Ibex 35	CVA6 35	Caliptra 16	Rocket 32	XiangShan 54	OpenTitan 245	Overall 417
Reference models							
Claude Opus 5 (Max)	88.6	91.4	**100.0**	**84.4**	**92.6**	82.0	**85.6**
GLM-5.3 (Max)	82.9	**97.1**	**100.0**	78.1	77.8	**82.4**	83.5
GPT-5.6 Sol (Max)	**91.4**	94.3	93.8	**84.4**	77.8	75.1	79.9
GPT-5.5 (xhigh)	88.6	94.3	93.8	71.9	79.6	71.0	76.5
DeepSeek-V4-Flash-0731	80.0	91.4	**100.0**	81.3	75.9	71.8	76.5
GLM-5.2 (Max)	77.1	88.6	93.8	65.6	75.9	70.2	73.6
Qwen3.6-27B	54.3	85.7	62.5	53.1	35.2	43.7	48.4
Base model: Qwen3.5-35B-A3B							
Qwen3.5-35B-A3B	**42.9**	71.4	43.8	28.1	**31.5**	29.0	34.5
LEGO-RL (OpenCode)	40.0	**74.3**	**56.3**	**37.5**	27.8	**35.9**	**39.3**
Base model: Qwen3.6-35B-A3B							
Qwen3.6-35B-A3B	62.9	68.6	37.5	25.0	27.8	36.3	39.3
KAT-Coder-V2.5-Dev	62.9	74.3	68.8	53.1	42.6	46.9	51.3
OriGent (SFT)	65.7	74.3	87.5	59.4	51.9	50.2	55.9
OriGent (SFT + RL)	**71.4**	**88.6**	**93.8**	**68.8**	**64.8**	**57.1**	**64.3**

Evaluation settings. The Qwen-based model comparisons use OpenCode with a 256K-token context window, a maximum of 64K generated tokens per response, and a sampling temperature of 1.0. Each HWE-Bench attempt has a 6-hour time limit; transfer benchmarks use their respective time limits. We report resolve rate, the percentage of tasks solved.

5.2 Main Results

Performance on HWE-Bench. SFT increases the resolve rate of Qwen3.6-35B-A3B from 39.3% to 55.9%, and online RL further raises it to 64.3% (Table 2). SFT improves performance across all six repositories, with the largest gains on Caliptra, from 37.5% to 87.5%, and Rocket, from 25.0% to 59.4%. Against KAT-Coder-V2.5-Dev, which shares the same base model, SFT alone achieves a 4.6-percentage-point improvement; this margin increases to 13.0 points after RL. OriGent also outperforms LEGO-RL (39.3%), which is based on Qwen3.5-35B-A3B. Claude Opus 5 achieves the highest overall resolve rate among the reference models at 85.6%.

Learning during RL. Figure 5 summarizes rollout statistics from 788 complete groups, comprising 6,304 trajectories before dynamic filtering. Mean reward varies between 0.38 and 0.58, with

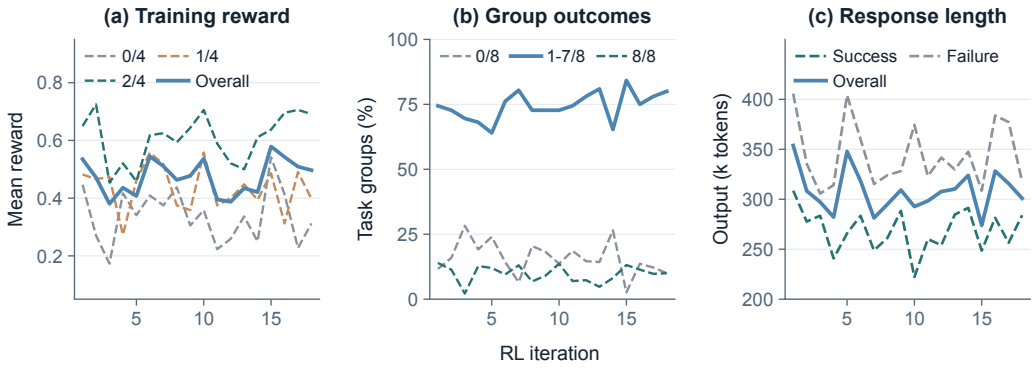


Figure 5: **RL training dynamics.** All panels use complete eight-trajectory groups before dynamic filtering. (a) Mean reward by fixed Qwen3.6-27B screening group and overall, weighted by trajectory counts. (b) Fractions of task groups with zero, one to seven, or eight successful trajectories. (c) Mean response length for successful and unsuccessful attempts and overall, measured as cumulative generated tokens per trajectory, including summaries.

tasks in the $2/4$ screening group consistently receiving higher rewards than those in the $0/4$ group (Figure 5a). Groups containing both successful and unsuccessful attempts account for 64.0–84.2% of complete groups in each iteration, providing group-relative learning signals throughout training (Figure 5b). Unsuccessful attempts generate more output tokens than successful ones in every iteration (Figure 5c). Across the observed trajectories, the 2,975 successful attempts average 269k output tokens, compared with 344k for the 3,329 unsuccessful attempts.

Transfer to software and terminal tasks. On SWE-bench Pro, SFT raises the resolve rate from 44.0% to 56.0%, and online RL further raises it to 58.0% (Table 3). Both exceed KAT-Coder-V2.5-Dev (48.5%) and LEGO-RL (43.5%) on the same 200-task subset. On Terminal-Bench 2.1, SFT resolves 34.8% of tasks, compared with 39.3% for the base. RL reaches 36.0%, corresponding to one additional solved task relative to SFT.

Table 3: **Transfer beyond hardware.** Resolve rates (%). Parentheses show percentage-point changes from Qwen3.6-35B-A3B for models based on it.

Model	SWE-bench Pro	Terminal-Bench 2.1
Tasks	200	89
KAT-Coder-V2.5-Dev	48.5 (+4.5)	**43.8** (+4.5)
LEGO-RL (OpenCode)	43.5	22.5
Qwen3.6-35B-A3B	44.0	39.3
+ SFT	56.0 (+12.0)	34.8 (−4.5)
OriGent (SFT + RL)	**58.0** (+14.0)	36.0 (−3.4)

The improvement on SWE-bench Pro alongside the decline on Terminal-Bench suggests that hardware-focused post-training transfers selectively across domains. A plausible explanation is that our training tasks reinforce repository-based problem solving, including code exploration, modification, and execution-based validation, which closely matches software repository tasks. This specialization may also shift the policy away from behaviors needed for broader terminal workflows, such as service configuration and file processing. Such a trade-off is consistent with prior findings that training in one domain can improve performance in some domains while reducing it in others (Li et al., 2025b).

5.3 Behavioral Changes after Post-Training

We compare interaction length and tool use on all 417 HWE-Bench tasks, with KAT-Coder-V2.5-Dev and GLM-5.3 as references. Turn, tool-call, and compaction counts are averaged over tasks,

with compaction summaries excluded from assistant turns. Total output includes reasoning and summaries; output per turn excludes summaries.

Table 4: **HWE-Bench trajectory statistics.** Interaction length and tool use across 417 tasks.

Metric	GLM-5.3	KAT	Base	SFT	SFT+RL
Output tokens, mean (k)	76.4	24.0	22.8	249.4	331.7
Output tokens, median (k)	71.3	16.8	12.9	173.7	178.3
Assistant turns	86.8	29.6	24.6	154.7	199.1
Output tokens per turn	880	808	925	1,454	1,503
Tool calls	97.5	92.4	31.2	176.0	220.7
Compactions	0.00	0.05	0.01	1.38	2.04

On HWE-Bench, the SFT model makes substantially longer attempts than the base model: median assistant turns rise from 18 to 116, and mean output grows from 22.8k to 249.4k tokens (Table 4).

One possible source of this difference is the length of the teacher demonstrations. In the SFT training set, successful trajectories have a median of 111 assistant turns and a 90th percentile of 219 (Appendix D.2). In a separate comparison on 1,060 synthesized RL tasks where both models succeed in the same environment, GLM-5.3 uses more turns, tool calls, and output tokens than the screening model Qwen3.6-27B (Table 5). Together with the demonstration lengths, this pattern suggests that the SFT model may inherit the teacher's extended interaction style alongside its task-solving behavior. Selecting demonstrations by final success retains the exploration and revisions preceding a solution, without identifying which steps could have been omitted.

Table 5: **Teacher and screening trajectories.** Successful attempts on 1,060 synthesized RL tasks. Values are means over tasks.

Metric	Qwen3.6-27B	GLM-5.3
Assistant turns	61.3	151.5
Output tokens (k)	24.5	97.2
Tool calls	73.0	162.3

RL further increases assistant turns by 28.7%, compared with 3.4% for output per turn. The test-based reward provides no explicit incentive to shorten an attempt, allowing additional interaction when pursuing a solution. Paired task outcomes show how this extra output relates to success: of the 66 tasks that change from unsuccessful under SFT to successful under RL, 28 use fewer output tokens. Conversely, the 129 tasks that remain unsuccessful account for 52.3% of the aggregate net increase in output tokens. Thus, some improvements accompany shorter solutions, while much of the additional output is spent on attempts that still fail.

KAT makes nearly three times as many tool calls as the base model with a similar amount of output. Under its larger context budget, GLM-5.3 solves more tasks with fewer turns and generated tokens than either SFT or RL. These comparisons highlight an efficiency gap in the current recipe. Controlling unproductive continuation and learning when to stop remain challenges beyond learning to complete tasks.

6 Conclusion

We present OriGent, a framework for synthesizing executable hardware environments and using them to post-train coding agents. The resulting collection of 18,257 tasks from 90 repositories supports both supervised fine-tuning and online reinforcement learning. OriGent reaches a resolve rate of 64.3% on HWE-Bench, improving over its base model by 25.0 percentage points. The gains extend to software engineering, with a resolve rate of 58.0% on a 200-task SWE-bench Pro subset, compared with 44.0% for the base model. For the hardware community, OriGent provides reusable environments and model baselines for studying agent post-training, together with a synthesis pipeline for extending task coverage to additional projects and workflows.

AI USE STATEMENT

We used LLM-based agents to construct task environments, tests, and problem statements, and to generate teacher trajectories, as described in Sections 3 and 4. Generated tests were checked on the base and repaired repository states, and teacher trajectories were filtered using execution outcomes. AI tools also assisted with code development, experimental design and analysis, and manuscript preparation. The authors reviewed and revised the manuscript and take responsibility for the methods, reported results, and conclusions.

REPRODUCIBILITY STATEMENT

Sections 3 and 4 describe environment synthesis and agent post-training. Appendix A documents repository selection, candidate filtering, and dataset composition. Appendices B and C detail the construction agents and execution checks and provide a complete task example. Training hyperparameters, supervision masks, and loss aggregation across context segments are specified in Appendix D, with evaluation settings in Section 5.1. We will open-source the synthesis pipeline code and release the synthesized task environments and trained model checkpoints.

REFERENCES

Manar Abdelatty, Maryam Nouh, and Sherief Reda. CovR: Coverage-aware hardware verification via reasoning-guided reinforcement learning. In *Proceedings of the ACM/IEEE International Symposium on Machine Learning for CAD*, 2026. URL https://arxiv.org/abs/2609.19189.

Anomaly. OpenCode: The open source coding agent. GitHub, 2026. URL https://github.com/anomalyco/opencode. Accessed September 22, 2026.

Anthropic. Introducing Claude Opus 5, 2026. URL https://www.anthropic.com/news/claude-opus-5.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Aleksandr Golubev. SWE-rebench V2: Language-agnostic SWE task collection at scale. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/63732.

Fan Cui, Chenyang Yin, Kexing Zhou, Youwei Xiao, Guangyu Sun, Qiang Xu, Qipeng Guo, Yun Liang, Xingcheng Zhang, Demin Song, and Dahua Lin. OriGen: Enhancing RTL code generation with code-to-code augmentation and self-reflection. In *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–9. ACM, 2024. doi: 10.1145/3676536.3676830. URL https://doi.org/10.1145/3676536.3676830.

Fan Cui, Hongyuan Hou, Zizhang Luo, Chenyun Yin, and Yun Liang. HWE-Bench: Benchmarking LLM agents on real-world hardware bug repair tasks. *arXiv preprint arXiv:2604.14709*, 2026. URL https://arxiv.org/abs/2604.14709.

DeepSeek-AI. DeepSeek-V4-Flash-0731 model card. Hugging Face, 2026. URL https://huggingface.co/deepseek-ai/DeepSeek-V4-Flash-0731.

Chenhui Deng, Yun-Da Tsai, Guan-Ting Liu, Zhongzhi Yu, and Haoxing Ren. ScaleRTL: Scaling LLMs with reasoning data and test-time compute for accurate RTL code generation. In *Proceedings of the ACM/IEEE Symposium on Machine Learning for CAD*, 2025. URL https://research.nvidia.com/labs/electronic-design-automation/publication/deng2025rtl/.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. SWE-Bench Pro: Can AI agents solve long-horizon software engineering tasks? In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/61047.

Yiming Du, Yuxin Jiang, Tao Yuan, Jianbo Dai, Shaowei Wang, Jierun Chen, Chaofan Tao, Xianzhi Yu, Lifeng Shang, Kam-Fai Wong, Xiaohui Li, and Haoli Bai. LEGO-RL: Harness-native reinforcement learning for coding agents. *arXiv preprint arXiv:2608.17393*, 2026. URL https://arxiv.org/abs/2608.17393.

Hamish Ivison, Junjie Oscar Yin, Rulin Shao, Teng Xiao, Nathan Lambert, and Hannaneh Hajishirzi. TMax: A simple recipe for terminal agents. *arXiv preprint arXiv:2606.23321*, 2026. URL https://arxiv.org/abs/2606.23321.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2E-Gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=7evvwwdo3z.

KwaiKAT Team. KAT-Coder-V2.5-Dev model card. Hugging Face, 2026. URL https://huggingface.co/Kwaipilot/KAT-Coder-V2.5-Dev.

Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo, Xiaohan Xu, Yue Zhang, Ziwei Tang, Yuanshuai Li, Florensia Widjaja, Xintong Zhu, Feige Zhou, Yongfeng Huang, Yannis Papakonstantinou, Fatma Ozcan, Chenhao Ma, and Reynold Cheng. SWE-SQL: Illuminating LLM pathways to solve user SQL issues in real-world applications. In *Advances in Neural Information Processing Systems*, volume 38, pp. 97085–97120, 2025a. doi: 10.52202/085713-3247. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8bfbf4ec87e1e331f0b1adc483b53b6b-Abstract-Conference.html.

Yu Li, Zhuoshi Pan, Honglin Lin, Mengyuan Sun, Conghui He, and Lijun Wu. Can one domain help others? a data-centric study on multi-domain reasoning via reinforcement learning. *arXiv preprint arXiv:2507.17512*, 2025b. URL https://arxiv.org/abs/2507.17512.

Yujiang Li, Zhenyu Hou, Yi Jing, Jie Tang, and Yuxiao Dong. CompactionRL: Reinforcement learning with context compaction for long-horizon agents. *arXiv preprint arXiv:2607.05378*, 2026. URL https://arxiv.org/abs/2607.05378.

Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. Invited paper: VerilogEval: Evaluating large language models for Verilog code generation. In *Proceedings of the IEEE/ACM International Conference on Computer Aided Design*, pp. 1–8, 2023. doi: 10.1109/ICCAD57390.2023.10323812. URL https://doi.org/10.1109/ICCAD57390.2023.10323812.

Mingjie Liu, Yun-Da Tsai, Wenfei Zhou, and Haoxing Ren. CraftRTL: High-quality synthetic data generation for Verilog code models with correct-by-construction non-textual representations and targeted code repair. In *International Conference on Learning Representations*, pp. 90377–90422, 2025a. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/e112a4671e8779aa9f640a0e3f81bd26-Abstract-Conference.html.

Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. RTLCoder: Fully open-source and efficient LLM-assisted RTL code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461, 2025b. doi: 10.1109/TCAD.2024.3483089. URL https://doi.org/10.1109/TCAD.2024.3483089.

Yi Liu, Changran Xu, Yunhao Zhou, Zeju Li, and Qiang Xu. DeepRTL: Bridging Verilog understanding and generation with a unified representation model. In *International Conference on Learning Representations*, pp. 93382–93404, 2025c. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/e9750610639c3e7a849cff746bf60dbd-Abstract-Conference.html.

Anton Lozhkov, Hugo Larcher, Mathieu Morlon, Loubna Ben Allal, and Leandro von Werra. The Stack v3: The largest open code dataset. Hugging Face, 2026. URL https://huggingface.co/datasets/HuggingFaceCode/stack-v3-train. Train-split language statistics, revision 8f3f25d86e44fd691428131efd17af75d4716499.

Miao Lu, Weiwei Sun, Weihua Du, Zhan Ling, Xuesong Yao, Kang Liu, and Jiecao Chen. Beyond the context window: Scaling agentic RL via end-to-end optimized context compression. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 21074–21125. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.966. URL https://aclanthology.org/2026.acl-long.966/.

Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design RTL generation with large language model. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference*, pp. 722–727, 2024. doi: 10.1109/ASP-DAC58780.2024.10473904. URL https://doi.org/10.1109/ASP-DAC58780.2024.10473904.

Wenhan Ma, Hailin Zhang, Liang Zhao, Yifan Song, Yudong Wang, Fuli Luo, and Zhifang Sui. Stabilizing MoE reinforcement learning by aligning training and inference routers. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/63177.

Mike Merrill, Alexander Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin, Jason Poulos, Maoyu Wang, Marianna Nezhurina, Di Lu, Orfeas Menis Mastromichalakis, Zhiwei Xu, Zizhao Chen, Yue Liu, Robert Zhang, Leon Liangyu Chen, Anurag Kashyap, Jan-Lucas Uslu, Jeffrey Li, Jianbo Wu, Minghao Yan, Song Bian, Vedang Sharma, Ke Sun, Steven Dillmann, Akshay Anand, Andrew Lanpouthakoun, Bardia Koopah, Changran Hu, Etash Guha, Gabriel Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong (Ryan) Wang, Changzhi Zhou, David Heineman, Hange Liu, Harsh Trivedi, John Yang, Junhong Lin, Manish Shetty, Michael Yang, Nabil Omi, Negin Raoof, Shanda Li, Terry Yue Zhuo, Wuwei Lin, Yiwei Dai, Yuxin Wang, Wenhao Chai, Shang Zhou, Dariush Wahdany, Ziyu She, Jiaming Hu, Zhikang Dong, Yuxuan Zhu, Sasha Cui, Ahson Saiyed, Arinbjörn Kolbeinsson, Christopher Rytting, Ryan Marten, Yixin Wang, Jenia Jitsev, Alex Dimakis, Andy Konwinski, and Ludwig Schmidt. Terminal-Bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 40903–40986, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/444a3737adaee10d86ad2ef5f74468e6-Paper-Conference.pdf.

Miles Team. Miles: Enterprise-grade reinforcement learning for large-scale model post-training. GitHub repository, 2026. URL https://github.com/radixark/miles.

OpenAI. GPT-5.5 system card, 2026a. URL https://deploymentsafety.openai.com/gpt-5-5.

OpenAI. GPT-5.6 system card, 2026b. URL https://deploymentsafety.openai.com/gpt-5-6.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with SWE-Gym. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 47717–47737. PMLR, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Nathaniel Pinckney, Chenhui Deng, Chia-Tung Ho, Yun-Da Tsai, Mingjie Liu, Wenfei Zhou, Brucek Khailany, and Haoxing Ren. Comprehensive Verilog design problems: A next-generation benchmark dataset for evaluating large language models and agents on RTL design and verification. *arXiv preprint arXiv:2506.14074*, 2025. URL https://arxiv.org/abs/2506.14074.

Qwen Team. Qwen3.5-35B-A3B model card. Hugging Face, 2026a. URL https://huggingface.co/Qwen/Qwen3.5-35B-A3B.

Qwen Team. Qwen3.6-35B-A3B model card. Hugging Face, 2026b. URL https://huggingface.co/Qwen/Qwen3.6-35B-A3B.

Qwen Team. Qwen3.6-27B model card. Hugging Face, 2026c. URL https://huggingface.co/Qwen/Qwen3.6-27B.

Yujiong Shen, Yajie Yang, Zhiheng Xi, Binze Hu, Huayu Sha, Qiyuan Peng, Jiazheng Zhang, Junlin Shang, Jixuan Huang, Yutao Fan, Jingqi Tong, Ming Zhang, Shihan Dou, Zhenfei Yin, Xingjun Ma, Lei Bai, Tao Gui, Qi Zhang, Xuanjing Huang, and Yu-Gang Jiang. SciAgentGym: Benchmarking multi-step scientific tool-use in LLM agents. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://icml.cc/virtual/2026/poster/66785.

Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Wayne Xin Zhao, Yang Song, Tao Zhang, and Ji-Rong Wen. SWE-Master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411*, 2026. URL https://arxiv.org/abs/2602.03411.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, Zhiming Mao, Xinyu Wang, Lifeng Shang, and Haoli Bai. SWE-Lego: Pushing the limits of supervised fine-tuning for software issue resolving. *arXiv preprint arXiv:2601.01426*, 2026. URL https://arxiv.org/abs/2601.01426.

Jing Wang, Shang Liu, Wenji Fang, Yuchao Wu, Yugao Zhu, and Zhiyao Xie. RTL-BenchLS: A large-scale benchmark for RTL reasoning and generation with large language models. *arXiv preprint arXiv:2606.08976*, 2026a. URL https://arxiv.org/abs/2606.08976.

Mingjun Wang, Yihan Wen, Yuntao Lu, Fengrui Liu, Yuxiang Zhao, Boyu Han, Jianan Mu, Yibo Lin, Runsheng Wang, Bei Yu, and Huawei Li. CircuitNet 3.0: A multi-modal dataset with task-oriented augmentation for AI-driven circuit design. In *International Conference on Learning Representations*, pp. 27308–27342, 2026b. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/2e0f5561c1553a97cee5fa64575358c9-Abstract-Conference.html.

Ran Xu, Yuchen Zhuang, Yishan Zhong, Yue Yu, Zifeng Wang, Xiangru Tang, Hang Wu, May Dongmei Wang, Peifeng Ruan, Donghan Yang, Tao Wang, Guanghua Xiao, Xin Liu, Carl Yang, Yang Xie, and Wenqi Shi. MedAgentGym: A scalable agentic training environment for code-centric reasoning in biomedical data science. In *International Conference on Learning Representations*, pp. 89688–89730, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/hash/911ca386fecaf7ebac35bb64733050fc-Abstract-Conference.html.

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 *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-3239. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Junyao Yang, Yucheng Shi, Zhongzhi Li, Ruhan Wang, Zongxia Li, Haitao Mi, and Leowei Liang. T1: Terminal agent reinforcement learning for long-horizon tasks. *arXiv preprint arXiv:2609.11042*, 2026. URL https://arxiv.org/abs/2609.11042.

Bowen Ye, Lei Li, Shicheng Li, Zihao Yue, Linghao Zhang, Hanglong Lv, Yuanxin Liu, Wenhan Ma, Hao Tian, Rang Li, Jinhao Dong, Yikai Zhao, Xiangwei Deng, Hailin Zhang, Liang Zhao, Qi Liu, Lingpeng Kong, Tong Yang, and Fuli Luo. CodeMidas: Scaling agentic coding RL environments from code itself. *arXiv preprint arXiv:2609.22068*, 2026. URL https://arxiv.org/abs/2609.22068.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Juncai Liu, Lingjun Liu, Xin Liu, Haibin Lin, Zhiqi Lin, Bole Ma, Guangming Sheng, Yuxuan Tong, Chi Zhang, Mofan Zhang, Ru Zhang, Wang Zhang, Hang Zhu, Jinhua Zhu, Jiaze Chen, Jiangjie Chen, Chengyi Wang, Hongli Yu, Yuxuan Song, Xiangpeng Wei, Hao Zhou, Jingjing Liu, Wei-Ying Ma, Ya-Qin Zhang, Lin Yan, Yonghui Wu, and Mingxuan Wang. DAPO: An open-source LLM reinforcement learning system at scale. In *Advances in Neural Information Processing Systems*, volume 38, pp. 113222–113244, 2025. doi: 10.52202/085713-3775. URL https://proceedings.neurips.cc/paper_files/paper/2025/file/a4277440d50f1f15d2cb4c14f7e0c0d2-Paper-Conference.pdf.

Zhongkai Yu, Yichen Lin, Chenyang Zhou, Yuwei Zhang, Kun Zhou, Junxia Cui, Haotian Ye, Zhengding Hu, Zaifeng Pan, Ruiyi Wang, Yujie Zhao, Hejia Zhang, Jingbo Shang, Jishen Zhao, and Yufei Ding. ChipMATE: Multi-agent training via reinforcement learning for enhanced RTL generation. *arXiv preprint arXiv:2605.12857*, 2026a. URL https://arxiv.org/abs/2605.12857.

Zhongkai Yu, Chenyang Zhou, Yichen Lin, Hejia Zhang, Haotian Ye, Junxia Cui, Zaifeng Pan, Jishen Zhao, and Yufei Ding. ChipBench: A next-step benchmark for evaluating LLM performance in AI-aided chip design. *arXiv preprint arXiv:2601.21448*, 2026b. URL https://arxiv.org/abs/2601.21448.

Z.ai. GLM-5.2 model card. Hugging Face, 2026a. URL https://huggingface.co/zai-org/GLM-5.2.

Z.ai. GLM-5.3 model card. Hugging Face, 2026b. URL https://huggingface.co/zai-org/GLM-5.3.

Hejia Zhang, Zhongming Yu, Chia-Tung Ho, Haoxing Ren, Brucek Khailany, and Jishen Zhao. LLM4Cov: Execution-aware agentic learning for high-coverage testbench generation. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026a. URL https://icml.cc/virtual/2026/poster/61436.

Shaoqiu Zhang, Maoquan Wang, Yuling Shi, Yuhang Wang, Xiaodong Gu, Yongqiang Yao, Rao Fu, and Shengyu Fu. FastContext: Training efficient repository explorer for coding agents. *arXiv preprint arXiv:2606.14066*, 2026b. URL https://arxiv.org/abs/2606.14066v1.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Xin Zhao, Ruihua Song, Yuan Zhang, Peng Wang, Cheng Chen, Jirong Wen, and Kai Jia. Immersion in the GitHub universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*, 2026a. URL https://arxiv.org/abs/2602.09892.

Yujie Zhao, Zhijing Wu, Boqin Yuan, Zhongming Yu, Hejia Zhang, Wentao Ni, Chia-Tung Ho, Haoxing Ren, and Jishen Zhao. PRO-V-R1: Reasoning enhanced programming agent for RTL verification. In *Proceedings of the 63rd ACM/IEEE Design Automation Conference*, 2026b. URL https://arxiv.org/abs/2506.12200v4.

Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. Group sequence policy optimization. *arXiv preprint arXiv:2507.18071*, 2025. URL https://arxiv.org/abs/2507.18071.

Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. QiMeng-CodeV-R1: Reasoning-enhanced Verilog generation. In *Advances in Neural Information Processing Systems*, volume 38, pp. 154266–154300, 2025. doi: 10.52202/085713-5160. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/e2f7407b62f152b7fe533fbc077fddb7-Abstract-Conference.html.

A Dataset Collection and Analysis

A.1 Candidate Collection and Filtering

We collected 257,099 pull requests (PRs) from 150 public GitHub repositories. Repository discovery combined 83 seed hardware projects with six initial GitHub queries. The queries covered four topics (`riscv`, `fpga`, `rtl`, and `cocotb`) and two languages (`Verilog` and `SystemVerilog`). They omitted forks and archived repositories and required more than 300 stars for `riscv`, `fpga`, and `Verilog`, 100 for `rtl`, 80 for `SystemVerilog`, and 50 for `cocotb`. HWE-Bench's six source repositories were removed before PR collection. The first two batches comprised 55 and 34 repositories from the query results; the second required at least 100 historical PRs. A later search added 61 repositories, including HDL tools, FPGA toolchains, firmware, and hardware emulators.

Table 6 lists the PR filters and retained counts. The full PR diff serves as the reference change; linked issues and tests are optional. GLM-5.2 removes PRs limited to documentation, maintenance or build changes, generated files, or structural edits without behavioral changes.

GPT-5.5 scores the remaining candidates from 0 to 2 on task value, behavior clarity, test signal, scope complexity, and interface clarity. Task value 2 denotes a substantive bug fix or behavior change, such as a correction to state-machine logic, bus protocols, or register configuration. Behavior clarity assesses whether the expected behavior can be recovered from the PR and code; a score of 1 allows some inference. A test-signal score of at least 1 requires identifiable, checkable behavior, such as a protocol response or register value, even when the PR includes no tests. A scope-complexity score of at least 1 requires a logic change beyond trivial or mechanical edits. Interface clarity assesses whether relevant entry points, such as APIs or commands, can be identified; a score of 1 allows some interface details to be inferred. We retain candidates with task value 2 and all other scores at least 1. PRs dominated by maintenance or build changes, generated or vendor updates, or repository synchronization are excluded, as are those requiring physical hardware for validation or having an unclear interface.

Keeping repositories with at least 20 such candidates leaves 35,173 PRs from 91 repositories. We defer Zephyr and its 10,170 candidates, leaving 25,003 PRs from 90 repositories for environment construction.

Table 6: Candidate filtering criteria and retained PR counts.

Stage	Filtering criterion	Retained PRs
Merge status	Retain merged pull requests.	211,829
Merge date	From 1 January 2017 through 29 May 2026.	205,563
Patch availability	Successfully retrieved, nonempty reference patch.	205,203
Patch size	20–2,000 added plus deleted lines across at most 50 files.	106,251
Behavior screening	Changes to hardware or software behavior.	74,034
Quality scoring	Task value 2; other scores ≥ 1; exclusion tags absent.	35,504 (134 repos)
Repository selection	At least 20 retained candidates per repository.	35,173 (91 repos)
Construction scope	Exclude the deferred Zephyr candidates.	25,003 (90 repos)

A.2 Dataset Characteristics

The task collection contains 18,257 tasks from 90 repositories. The SFT training set uses 8,013 tasks from 61 of these repositories, while the RL training set uses 2,163 tasks from 23 other repositories.

Tasks are grouped by the primary engineering role of their source repository. Simulation and verification covers HDL simulators, verification frameworks, and architecture or instruction-set emulators. Firmware and embedded software also includes device runtimes, drivers, and host-side embedded development tools.

Changes across files and languages. Patch size is measured by summing added and deleted lines in the complete reference diff, including changes to tests, configuration, and documentation, as well as comments and blank lines. File counts include binary files, which do not contribute to line counts. Across the collection, 89.1% of reference patches modify at least two files. This remains 84.6% when counting only files with recognized code extensions, including scripts and intermediate representations. We identify languages and code representations by file extension; 36.6% of reference patches span at least two of these categories. Such changes are most prevalent in EDA software (61.8%) and simulation and verification (56.4%), compared with 6.1% in firmware and embedded software. Cross-language changes are thus concentrated in the toolchain and verification parts of the collection.

Languages across engineering areas. Figure 6 groups reference patches by their dominant language, defined as the language accounting for the most added and deleted lines among recognized files. C/C++ is dominant in 77.8% of firmware and embedded software patches and 68.7% of EDA patches. Rust accounts for a further 16.4% in firmware and embedded software, while MLIR accounts for 11.8% in EDA. Hardware DSLs and generators have a more even mix of Scala, Python, Haskell, and Rust, at 24.2%, 23.3%, 21.3%, and 19.5%, respectively. The collection therefore covers the implementation languages of hardware tools and system software alongside HDLs.

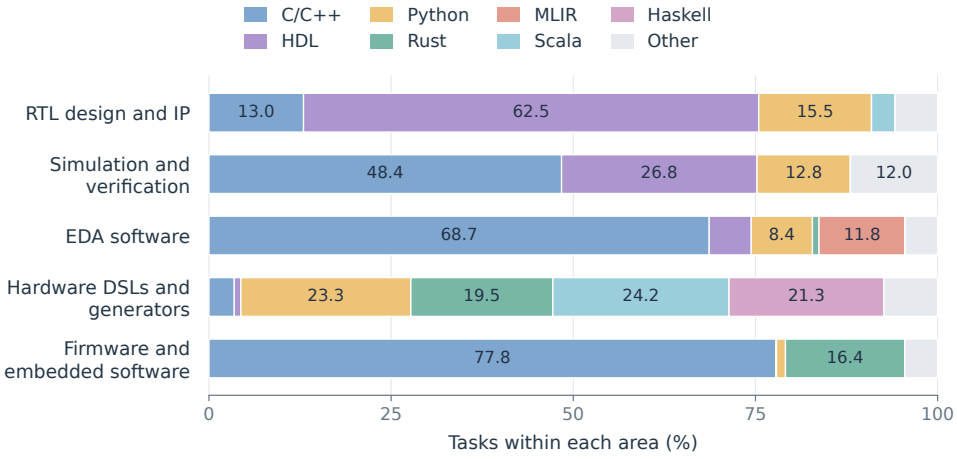


Figure 6: **Dominant language of reference patches by repository area.** Percentages are normalized within each area. HDL combines Verilog, SystemVerilog, and VHDL. Other includes remaining languages, ties, and patches without a recognized code extension.

Table 7 lists all 90 source repositories and the number of tasks from each that enter SFT and RL. CIRCT, NuttX, and XRT together account for 34.8% of all tasks.

Table 7: Complete repository inventory. Counts refer to unique tasks in the task collection (All tasks) and the SFT and RL subsets. Area labels are RTL design and IP (RTL), simulation and verification (Sim.), EDA software (EDA), hardware DSLs and generators (DSL), and firmware and embedded software (FW).

Repository	Area	All tasks	SFT	RL
`amaranth-lang/amaranth`	DSL	128	0	29
`analogdevicesinc/hdl`	RTL	175	0	35
`antonblanchard/microwatt`	RTL	43	41	0
`apache/nuttx`	FW	2,267	0	616
`bespoke-silicon-group/basejump_stl`	RTL	74	69	0
`black-parrot/black-parrot`	RTL	163	0	46
`cartesi/machine-emulator`	Sim.	20	16	0
`chipsalliance/caliptra-mcu-sw`	FW	221	0	41

Continued on next page

Table 7 (continued)

Repository	Area	All tasks	SFT	RL
chipsalliance/chisel	DSL	281	211	0
chipsalliance/i3c-core	RTL	22	20	0
chipsalliance/surelog	EDA	566	467	0
chipsalliance/uhdm	EDA	147	138	0
chipsalliance/verible	EDA	64	0	0
clash-lang/clash-compiler	DSL	353	0	143
cocotb/cocotb	Sim.	152	0	36
cornell-zhang/allo	DSL	46	44	0
earlephilhower/arduino-pico	FW	120	0	36
efabless/openlane2	EDA	17	0	0
embassy-rs/embassy	FW	306	0	59
enjoy-digital/litedram	RTL	45	37	0
enjoy-digital/liteeth	RTL	22	13	0
enjoy-digital/litex	RTL	231	0	48
esp-rs/esp-hal	FW	41	0	2
f4pga/f4pga-arch-defs	EDA	126	0	41
f4pga/prjxray	EDA	58	46	0
fastmachinelearning/hls4ml	DSL	172	79	0
ferrandi/panda-bambu	EDA	4	1	0
fpga-research/fabulous	EDA	33	21	0
freertos/freertos-kernel	FW	121	95	0
gem5/gem5	Sim.	329	0	122
ghdl/ghdl	Sim.	37	31	0
glasgowembedded/glasgow	FW	15	10	0
google/xls	EDA	74	0	0
greatscottgadgets/luna	RTL	20	18	0
intel/rohd	DSL	98	94	0
lekkit/rvvm	Sim.	23	16	0
limine-bootloader/limine	FW	34	27	0
llvm/circt	EDA	2,619	2,222	0
lnis-uofu/openfpga	RTL	107	76	0
logisim-evolution/logisim-evolution	Sim.	105	0	30
lowrisc/sonata-system	RTL	40	0	0
mikepopoloski/slang	EDA	145	128	0
modm-io/modm	FW	80	0	0
najaeda/naja	EDA	66	62	0
openhwgroup/core-v-verif	Sim.	89	71	0
openhwgroup/cv-hpdcache	RTL	24	20	0
openhwgroup/cv32e40p	RTL	87	77	0
openhwgroup/cv32e40s	RTL	58	57	0
openhwgroup/cv32e40x	RTL	134	117	0
openhwgroup/cvw	RTL	123	97	0
oreboot/oreboot	FW	29	23	0
pavel-demin/red-pitaya-notes	RTL	28	26	0
probe-rs/probe-rs	FW	385	0	100
pulp-platform/ara	RTL	53	33	0
pulp-platform/axi	RTL	38	32	0
pulp-platform/cheshire	RTL	20	15	0
pulp-platform/common_cells	RTL	26	21	0
pulp-platform/floonoc	RTL	15	11	0
pymtl/pymtl3	DSL	54	43	0
riscv-boom/riscv-boom	RTL	11	8	0
riscv-collab/riscv-openocd	FW	102	85	0
riscv-non-isa/riscv-arch-test	Sim.	52	44	0
riscv-software-src/opensbi	FW	18	17	0
riscv-software-src/riscv-isa-sim	Sim.	155	144	0
rt-thread/rt-thread	FW	534	0	132
rustsbi/rustsbi	FW	10	7	0
siliconcompiler/siliconcompiler	EDA	163	0	29
slaclab/surf	RTL	153	0	48
spinalhdl/spinalhdl	DSL	118	0	25

Continued on next page

Table 7 (continued)

Repository	Area	All tasks	SFT	RL
stnolting/neorv32	RTL	295	0	83
sysprog21/rv32emu	Sim.	70	51	0
the-openroad-project/openlane	EDA	14	0	0
the-openroad-project/openroad	EDA	772	676	0
trabucayre/openfpgaloader	FW	12	11	0
ucb-bar/chipyard	RTL	42	26	0
ucb-bar/gemmini	RTL	24	21	0
ucsbarchlab/pyrtl	DSL	39	29	0
verilator/verilator	Sim.	857	0	181
verilog-to-routing/vtr-verilog-to-routing	EDA	314	226	0
veryl-lang/veryl	DSL	353	307	0
vhdl-ls/rust_hdl	EDA	60	54	0
vunit/vunit	Sim.	42	39	0
xilinx/mlir-aie	EDA	350	241	0
xilinx/pynq	FW	53	47	0
xilinx/xrt	FW	1,469	1,194	0
yosyshq/apicula	EDA	73	51	0
yosyshq/nextpnr	EDA	272	0	120
yosyshq/sby	Sim.	26	23	0
yosyshq/yosys	EDA	705	0	161
zigembeddedgroup/microzig	FW	126	87	0
Total		18,257	8,013	2,163

B Construction Agents and Execution Checks

Models and execution frameworks. Repository profiles are prepared with GPT-5.6 Sol in Codex. Task construction uses GPT-5.6 Sol in Codex and, for later batches, GLM-5.3 in OpenCode 1.18.22. Problem-statement generation and review use GLM-5.3 in separate Codex or OpenCode sessions.

Profile preparation. The profile agent receives the repository, candidate-change metadata, and build files. It identifies the tools and dependencies needed by the candidate revisions and defines separate profiles where these requirements differ. Its outputs are a repository template describing the base image, tools, dependencies, and working directory, together with repository-specific guidance for task construction. A Dockerfile and setup script are generated from the template for each profile. Each profile is checked by building its image and running representative candidate tasks before reuse. Task agents flag failures caused by a shared toolchain configuration as requiring profile revision.

The 18,257 tasks use 152 repository-specific profiles, with a mean of 120.1 tasks per profile and a median of 38.5. Of the 90 repositories, 53 use a single profile.

Task construction. For each candidate, the task agent receives the base commit, reference patch, source metadata, repository template, and repository-specific guidance. It selects an existing profile and writes three scripts: `prepare_script.sh` for the task-specific setup, `test.sh` for the target behavior, and `regression.sh` for related behavior to preserve. The test scripts create any temporary testbenches or stimuli they require. The agent invokes the executor during construction and revises its scripts in response to build and test output.

Execution checks. Target and regression tests are run on both the base and repaired states, each in a fresh container created from the same prepared image. The reference patch is applied only for the repaired-state checks. A base-state target check is accepted when it exits nonzero and its log contains the designated failure marker from the behavioral assertion. The repaired-state target check must exit zero and emit its success marker; both regression checks must exit zero. Timeouts and build failures do not satisfy the target-failure condition. Appendix C gives an example of the scripts and their recorded outcomes.

Problem-statement generation and review. After execution validation, a generation agent drafts the problem statement from the original issue or PR. It consults the base-state code, reference patch, tests, and execution evidence to fill in missing behavioral and public interface requirements. The statement describes the observed failure and expected behavior while leaving implementation choices open. It excludes reference patch excerpts and repair instructions. A separate review agent first reads the source material and execution evidence to establish the verified behavior, then checks the proposed statement for completeness and ambiguity. The review produces an `approved`, `revised`, or `rejected` decision. Approved and revised statements are retained; rejected statements are excluded.

C Example of a Constructed Task

This task, derived from LUNA PR 113, concerns the constant-stream generator used in USB gateware. When transmission starts at a nonzero byte offset, the first payload incorrectly comes from offset zero. The task uses the `legacy-nmigen-py39` profile, which prepares Python 3.9 and the historical nMigen dependencies. Both tests simulate the hardware through LUNA's test harness in the USB clock domain. The excerpts below show the stimuli and checks, omitting test-runner setup and diagnostic logging.

Problem statement. The following is the generated statement used for this task.

> ConstantStreamGenerator does not emit the requested byte when it is started with a nonzero start_position. For example, with constant_data = b"HELLO, WORLD", start_position = 2, and a positive max_length, the first beat after start is asserted currently presents H, the byte at offset 0. Instead, when that first beat is presented, stream.valid and stream.first must be asserted and stream.payload must equal constant_data[2], i.e. L. More generally, the first payload emitted after a start strobe must be the byte at the supplied start_position, not the byte at offset 0.
>
> The existing zero-offset behavior must remain unchanged: after start_position = 0 and a positive max_length, stream.valid and stream.first are asserted with stream.payload equal to constant_data[0]; that byte remains presented while stream.ready is low, and stream.payload advances to constant_data[1] after stream.ready is asserted. This affects the byte-oriented constant-stream feature used for USB descriptor responses, including descriptor data resumed at a nonzero position.

Target test. The test configures `ConstantStreamGenerator` with `b"HELLO, WORLD"` and starts transmission at offset 2. It checks the first payload and its valid/first indicators; the expected byte is `L`.

```
yield dut.start_position.eq(2)
yield dut.max_length.eq(1000)
yield from self.advance_cycles(4)
yield from self.pulse(dut.start)

valid = yield dut.stream.valid
first = yield dut.stream.first
actual = yield dut.stream.payload
expected = ord("L")

if valid != 1 or first != 1 or actual != expected:
    self.fail(
        f"first valid payload at offset 2: valid={valid}, "
        f"first={first}, expected={expected}, actual={actual}"
    )
```

Regression test. Using the same constant data, the regression test starts at offset 0 and holds `stream.ready` low for three cycles. The first byte must remain `H` until the receiver becomes ready, after which the stream advances to `E`.

```
yield dut.start_position.eq(0)
yield dut.max_length.eq(5)
yield dut.stream.ready.eq(0)
yield from self.advance_cycles(4)
yield from self.pulse(dut.start)

self.assertEqual((yield dut.stream.valid), 1)
self.assertEqual((yield dut.stream.first), 1)
self.assertEqual((yield dut.stream.payload), ord("H"))

yield from self.advance_cycles(3)
self.assertEqual((yield dut.stream.valid), 1)
self.assertEqual((yield dut.stream.first), 1)
self.assertEqual((yield dut.stream.payload), ord("H"))

yield dut.stream.ready.eq(1)
yield
yield
self.assertEqual((yield dut.stream.valid), 1)
self.assertEqual((yield dut.stream.first), 0)
self.assertEqual((yield dut.stream.payload), ord("E"))
```

Table 8 reports the recorded outcomes on the base and repaired states.

Table 8: Recorded execution checks for LUNA PR 113.

Repository state	Check	Exit code	Recorded outcome
Base	Target	1	First payload: `H`; expected `L`
Repaired	Target	0	First payload: `L`
Base	Regression	0	Hold `H`, then advance to `E`
Repaired	Regression	0	Hold `H`, then advance to `E`

D Training Settings

D.1 Training Configuration

Optimization and loss aggregation. Let K_i denote the number of segments with supervised tokens in trajectory i, and let $m_{ist} \in \{0, 1\}$ be the token mask for segment s. The supervised-token counts are $n_{is} = \sum_t m_{ist}$ for a segment and $N_i = \sum_{s=1}^{K_i} n_{is}$ for its original trajectory. The length-normalized likelihood ratio for each segment is

$$\rho_{is}(\theta) = \exp\left[\frac{1}{n_{is}} \sum_t m_{ist} \left(\log \pi_\theta(y_{ist} \mid c_{ist}) - b_{ist}\right)\right], \tag{1}$$

where c_{ist} is the recorded context for token y_{ist} and b_{ist} is the fixed log-probability baseline for the update. For a batch of B original trajectories, we minimize

$$\mathcal{L}_{\mathrm{GSPO}}(\theta) = -\frac{1}{B} \sum_{i=1}^{B} \sum_{s=1}^{K_i} \frac{n_{is}}{N_i} \min[\rho_{is} A_i,\ \mathrm{clip}(\rho_{is}, 1-\epsilon_l, 1+\epsilon_h) A_i], \tag{2}$$

where A_i is the advantage computed within the task's trajectory group, and ϵ_l and ϵ_h are the clipping thresholds.

Each RL batch receives one optimizer update. The denominator uses $b_{ist} = \mathrm{sg}[\log \pi_\theta(y_{ist} \mid c_{ist})]$ from the same training forward pass, where sg stops gradient propagation. Thus $\rho_{is} = 1$ in value and clipping is inactive for these updates, while gradients flow through the numerator. Rollout log-probabilities are retained for diagnostics.

Table 9: Training settings for the SFT and RL stages. TP, PP, CP, DP, EP, and ETP denote tensor, pipeline, context, data, expert, and expert tensor parallelism, respectively.

Setting	SFT	RL
H100 80GB GPUs	32	64
Training TP / PP / CP / DP	2 / 4 / 4 / 1	2 / 4 / 4 / 2
Training EP / ETP	8 / 1	8 / 1
Rollout TP / EP	—	8 / 8
Concurrent agent sandboxes	—	256
Training tasks	8,013	2,163
Training repositories	61	23
Batch size	32 segments	32 tasks $\times$ 8 rollouts
Epochs / updates per batch	3 epochs	1 update
Learning rate	$10^{-5} \rightarrow 10^{-6}$	10^{-6}
Schedule	Cosine; 5% warmup	Constant; no warmup
Optimizer	Adam	Adam
Gradient clipping	1.0	1.0
Context limit	262,144 tokens	262,144 tokens
Generation limit per response	—	65,536 tokens
Sampling temperature / top-p	—	1.0 / 1.0
GSPO clipping (ϵ_{low}, ϵ_{high})	—	3×10^{-4}, 4×10^{-4}
Reference-policy KL regularization	—	Disabled
Entropy coefficient	—	0
Compaction reserved tokens	—	73,728 (72K)
Working-context threshold	—	188,416 (184K) tokens
Agent / verifier time limit	—	3 hours / 1 hour

The implementation averages per-token policy losses over all supervised tokens of each original trajectory and then over trajectories, giving segment s weight n_{is}/N_i. The mask excludes prompt tokens, tool observations, and later copies of summaries used as context; summary generation is supervised under the admission rule below.

Context handling and replay. OpenCode reserves 72K tokens and triggers compaction at 184K tokens. Tool-output pruning is disabled. The agent time limit covers the complete attempt, including summary generation; final verification has a separate time budget.

Summaries generated during compaction contribute to the policy loss if generation completes normally and the nonempty summary text is used to continue the attempt. Later copies serve only as input.

Miles TITO records prompt and generated token IDs during rollout and uses them directly for training; tool observations provide context without contributing to the loss. R^3 replays the recorded expert selections for each token and MoE layer in the training forward pass, using the same segment boundaries as the token sequences.

Synchronous execution. Rollout collection dominates runtime: across six completed iterations on 64 GPUs, it accounts for 86.6% of framework-reported step time, compared with 9.6% for training and 3.8% for other operations. This imbalance favors allocating most GPUs to rollout in an asynchronous setup. Our GPU budget does not support such a split while meeting the memory requirements of long-context training. We therefore adopt synchronous training, which also avoids policy staleness.

D.2 SFT Data and Supervision Statistics

The final SFT set contains 8,013 trajectories and 8,992 training segments. Each epoch processes 1.20B tokens, of which 441.6M are supervised. Tables 10 and 11 summarize turn masking and sequence lengths, respectively.

Table 10: Assistant-turn masking in the final SFT set. Counts exclude repeated context and compaction summaries. Each masked turn is counted once, in the order shown.

Category	Assistant turns
Before turn masking	963,088
Tool-invocation error	15,780
Incomplete response	3
Empty reasoning	447,261
Reasoning longer than 16,384 tokens	1,457
Retained for supervision	498,587

Table 11: Length distributions in the final SFT set.

Quantity	Median	90th	95th	99th	Maximum
Assistant turns per trajectory	111	219	248	286	300
Input tokens per segment	129,157	236,942	243,239	250,796	262,114
Supervised tokens per segment	45,853	93,360	106,038	125,476	159,432

D.3 RL Task Selection

Table 12 summarizes the RL task pool after screening with Qwen3.6-27B. Tasks in the $0/4$ group are retained only if GLM-5.3 solves them in the same environment version.

Table 12: RL task selection. Columns $0/4$, $1/4$, and $2/4$ count retained tasks by screening outcome.

Repositories	Candidates	$0/4$	$1/4$	$2/4$	Retained
23	8,199	705	649	809	2,163

D.4 Training–Rollout Consistency

To monitor numerical differences between rollout and training, we compare the log-probabilities assigned by SGLang and Megatron to the same generated tokens before each policy update. Absolute differences are averaged over supervised tokens within each original trajectory and then across the 256 trajectories in the batch. Across updates 2–18 with rollout routing replay enabled, the mean absolute difference remains near 0.02, ranging from 0.0187 to 0.0253 (Figure 7).

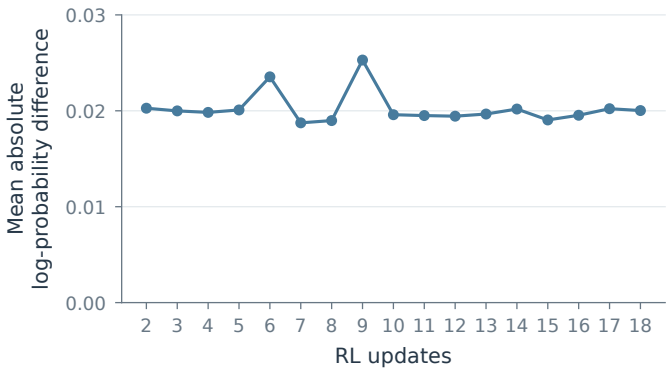


Figure 7: **Training–rollout log-probability difference.** Mean absolute differences for completed RL updates 2–18 with routing replay.

D.5 Evaluation Trajectory Statistics

Table 4 summarizes trajectories from all 417 HWE-Bench tasks, with one recorded trajectory for each model–task pair. KAT denotes KAT-Coder-V2.5-Dev. The Qwen-based models use a 256K context window and a 64K output limit per response; GLM-5.3 uses a 1M context window, a 128K output limit, and tool-output pruning. All have a six-hour agent budget.

Turn, tool-call, and compaction counts are averaged over tasks; output length is reported as both a mean and a median. Output tokens include reasoning and compaction summaries, excluding input context and tool observations. Assistant turns count assistant responses other than compaction summaries. Output tokens per turn are computed by dividing recorded non-compaction output by the number of assistant turns. Compactions count generated context summaries.

D.6 Teacher and Screening Trajectory Comparison

Table 5 compares GLM-5.3 teacher trajectories with Qwen3.6-27B screening trajectories on synthesized tasks selected for RL. Both models must have a successful attempt in the same prepared environment. After teacher-trajectory quality filtering, the comparison covers 1,060 tasks: 447 from the $1/4$ screening group and 613 from the $2/4$ group. For each task, we average the one or two successful Qwen attempts before averaging over tasks; GLM contributes one successful trajectory per task. This gives 1,673 Qwen attempts and 1,060 GLM attempts.

GLM uses more turns than the mean successful Qwen attempt on 1,009 of the 1,060 tasks (95.2%). The difference appears in both screening groups: Qwen and GLM average 68.8 and 164.9 turns in the $1/4$ group, and 55.8 and 141.8 in the $2/4$ group.