CODEMIDAS: SCALING AGENTIC CODING RL ENVIRONMENTS FROM CODE ITSELF

Anonymous authors
Paper under double-blind review

ABSTRACT

Training capable coding agents via reinforcement learning (RL) requires diverse tasks with reliable verifiers. Open-source codebases offer a rich source of such tasks, while existing methods typically rely on development artifacts such as issues and commits, limiting the range of tasks that can be extracted. To better scale RL environments, we present CodeMidas, an agentic pipeline that turns implemented functionality in existing codebases into executable RL environments using source code as its only task-specific input. CodeMidas allocates agentic compute to every stage of environment construction: agents explore implemented functionality to formulate behavioral specifications, construct tests grounded in execution of the original code, and validate and filter candidate tasks through execution checks and repeated solution rollouts. The resulting dataset has 5,545 training tasks from 3,185 open-source codebases spanning 23 programming languages and 15 technical domains. Training MiMo-V2.5 on these tasks with GRPO improves performance on all five diverse benchmarks, covering issue repair (DeepSWE + 11.7%), whole-program construction (ProgramBench +17%), and terminal work (Terminal-Bench v2.1 +8.5%). Ablations show that increasing the number of high-quality training tasks improves performance. Trajectory analysis shows the RL-trained agent demonstrates better behaviors like increasing codebase exploration and more diverse self-verification. These results establish source code as a scalable foundation for constructing RL environments that improve coding agents.

1 INTRODUCTION

Large language models are increasingly capable of agentic coding: completing substantial pieces of real software work autonomously over long horizons (Yang et al., 2024; Wang et al., 2025a). Building on reinforcement learning (RL) for code generation (Le et al., 2022; Zeng et al., 2025), recent studies have shown substantial gains in real-world software engineering (Wei et al., 2025). Effective RL requires diverse tasks to support generalization and reliable rewards to reinforce correct work (Yang et al., 2025). A central challenge is thus to turn real codebases into a broad supply of training tasks.

Existing pipelines for coding RL commonly construct tasks from development artifacts. Some derive task statements from issues, pull requests, or commits (Jimenez et al., 2024; Jain et al., 2025; Chen et al., 2026a). Others synthesize tasks around existing tests or use documentation to specify the requests (Jain et al., 2024; Zhang et al., 2025; Zeng et al., 2026). These approaches tie task creation to the coverage of these artifacts. This motivates constructing RL environments directly from code and establishing their training value for long-horizon agentic RL.

Open-source codebases provide a rich foundation for this approach, with large code corpora covering hundreds of programming languages (Lozhkov et al., 2024). Their public interfaces and observable behavior help define task requirements, while executing the original code informs test expectations. The surrounding code can be adapted into real development environments. Realizing this potential requires making the required behavior explicit in the task statement while leaving internal implementation choices open (Badertdinov et al., 2026) with verifiers that reject incorrect solutions while accepting alternative correct implementations (Barr et al., 2015; Liu et al., 2023; Wang et al., 2026).

We present **CodeMidas**, an agentic pipeline that constructs executable coding RL environments using source code as its only task-specific input. It allocates agentic compute throughout construction: LLM agents use the tools of a coding harness, such as Claude Code (Anthropic, 2025), to explore

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: Task-specific inputs. ✓ = not required; ♦ = required by some variants; ✗ = required. Descriptions: human-written docs or docstrings, not issue/PR text. Scope: one function (Func.) or codebase.

Method	w/o issue	w/o PR	w/o commit	w/o existing tests	w/o written description	Scope	Lang.
SWE-rb V2 (Badertdinov et al., 2026)	♦	✗	✗	✗	✓	Codebase	20
SWE-Universe (Chen et al., 2026a)	✗	✗	✗	✗	✓	Codebase	9
Multi-SWE-RL (Zan et al., 2025)	✗	✗	✗	✗	✓	Codebase	7
R2E-Gym (Jain et al., 2025)	✓	✓	✗	♦	✓	Codebase	1
SSR (Wei et al., 2026)	✓	✓	♦	✗	✓	Codebase	NA
SWE-Dev (Du et al., 2025)	✓	✓	✓	✗	✗	Codebase	1
RepoST (Xie et al., 2025)	✓	✓	✓	✓	✓	Func.	1
CodeMidas	✓	✓	✓	✓	✓	**Codebase**	**23**

codebases and formulate behavioral specifications, adapt codebases into development starting points, and construct tests grounded in execution of the original code. CodeMidas then applies post-rollout filtering: adversarial rollouts probe for exploitable leakage, solution reviews assess verifier decisions against the stated requirements, and rollout success rates guide task selection.

Using CodeMidas, we construct 5,545 verifiable training tasks from 3,185 open-source codebases spanning 23 programming languages and 15 technical domains. Training MiMo-V2.5 (Xiaomi MiMo Team, 2026b) on these tasks with GRPO (Shao et al., 2024) improves performance on all five external benchmarks, spanning issue repair, whole-program construction, code translation, and terminal work. Gains over the initial policy include 11.7 points on DeepSWE (Huang & Jiang, 2026), 17.0 points in the Almost Resolved score on ProgramBench (Yang et al., 2026), and 8.5 points on Terminal-Bench v2.1 (Merrill et al., 2026). With the same initial policy and training configuration, RL on tasks produced by CodeMidas' full pipeline outperforms RL on over 30,000 tasks from public PR-derived datasets across all five benchmarks. Training on the same tasks also improves the smaller MiMo-V2.6-Distill-Qwen-9B model (Xiaomi MiMo Team, 2026a) on four external benchmarks.

We further examine (a) how task supply can expand, (b) how task scale and quality affect these gains, and (c) how agent behavior evolves during training. We find that (a) task supply can expand beyond development artifacts and grow through continued exploration and broader codebase coverage; (b) more high-quality tasks yield progressively higher scores on SWE-bench Pro (Deng et al., 2026), DeepSWE, and CodeMidas Val, and even the 3k subset outperforms a vanilla 8k sample drawn before cleaning and filtering; and (c) as RL progresses, agents explore codebases more and perform more varied self-verification, with agent-written checks associated with higher success rates. These changes also appear on external tasks, providing behavioral evidence of generalization.

Together, these findings point to source code itself as a basis for scaling coding RL: data analyses show how implemented functionality broadens task supply beyond development-artifact boundaries, and RL experiments demonstrate the training value of environments produced by CodeMidas's complete construction and filtering pipeline across diverse forms of software work.

2 Related Work

CodeMidas builds upon recent efforts to scale tasks and verification for coding RL.

Scalable RL Environment Construction Following SWE-bench (Jimenez et al., 2024), PR-based pipelines (Badertdinov et al., 2026; Chen et al., 2026a; Zan et al., 2025) construct executable RL environments; others (Jain et al., 2025; Luo et al., 2025) derive tasks from commits. Codebase tests support implementation tasks (Du et al., 2025) and policy-guided generation (Wei et al., 2026; Xiao et al., 2026; Kim et al., 2026). Task supply in these pipelines follows the coverage of development records or existing tests. Evaluation and supervised-training pipelines use tests (Yang et al., 2025; Zhang et al., 2025; Zeng et al., 2026), documentation (Chen et al., 2026b), or development artifacts (Fu et al., 2026; Zhao et al., 2026; Liang et al., 2026; Zhang et al., 2026a); RepoST (Xie et al., 2025) also starts from source code but builds single-function Python tasks for fine-tuning. CodeMidas uses source code as its only task-specific input (Table 1) to build codebase-level environments across 23 languages for long-horizon agentic RL with training-time reward audits.

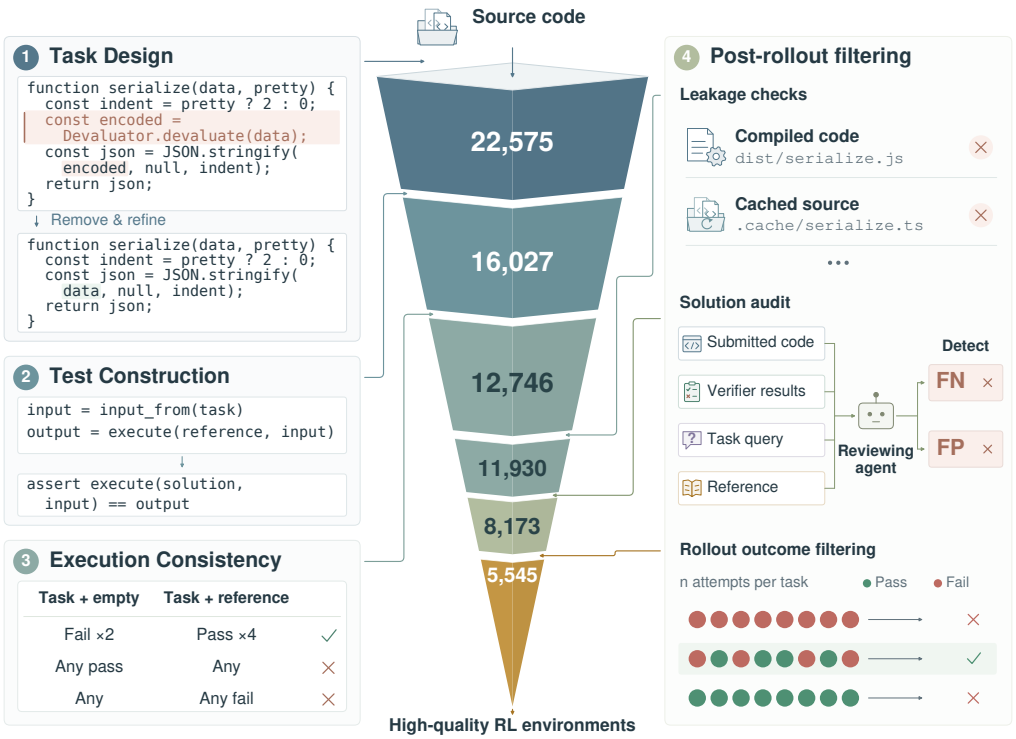

Figure 1: CodeMidas overview. Four modules cover task design, test construction, execution consistency, and post-rollout filtering. The pyramid shows retained task counts respectively.

Verification for Coding RL Coding RL uses test-based, learned, and patch-similarity rewards (Le et al., 2022; Zeng et al., 2025; Wei et al., 2025). Verification supports solution selection (Chen et al., 2023; Jain et al., 2024), action or patch scoring (Dihan & Khan, 2026; Raghavendra et al., 2026), and refinement (Chen et al., 2024; Shinn et al., 2023). Test-based rewards benefit from stronger tests (He et al., 2025; Wang et al., 2025b), yet remain vulnerable to false positives (Barr et al., 2015; Liu et al., 2023; Wang et al., 2026), unsupported rejections (Deng et al., 2026; Badertdinov et al., 2026), and harness exploits (Baker et al., 2025). CodeMidas grounds tests in reference execution, checks consistency, and audits rollouts for leakage and verifier defects before RL.

3 Method

This section describes how CodeMidas constructs and filters coding RL environments (Figure 1): task design (§3.1), test construction (§3.2), environment preparation (§3.3), post-rollout filtering (§3.4), and the resulting dataset (§3.5). Throughout the pipeline, agents are LLMs that use the tools of a coding harness to read, edit, and execute code; unless otherwise stated, we use MiMo-V2.5-Pro (Xiaomi MiMo Team, 2026b) in Claude Code (Anthropic, 2025) for its favorable trade-off between cost and performance. Appendix A reports stage-wise attrition and failure cases.

3.1 Task Design and Codebase Adaptation

We use agents to inspect codebase structure and build metadata to identify candidate functionality with public entry points and observable outcomes. Supported interfaces include command-line tools, pure library functions, and stateful library APIs, assessed through process outputs, return values, and state changes across calls. We prioritize tasks requiring reasoning across the codebase and consider up to five candidates per codebase when constructing the dataset.

For each candidate, we trace public entry points and shared dependencies to define the task scope. The task statement specifies inputs, observable behavior, and required public interfaces, leaving internal helpers and algorithms to the solver. We revise the statement and code boundaries together to establish

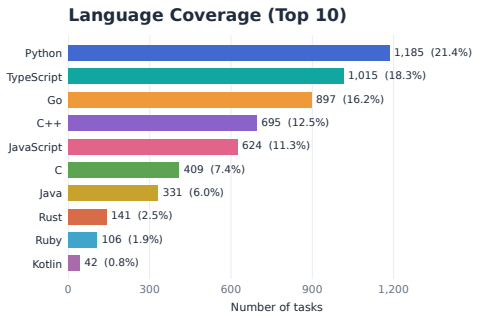

Figure 2: Language coverage of the training set. Tasks inherit the primary language label of their codebase. The ten most frequent languages cover 5,445 of 5,545 tasks (98.2%); the full dataset spans 23 languages. Labels report task counts and the associated percentages of the full training set.

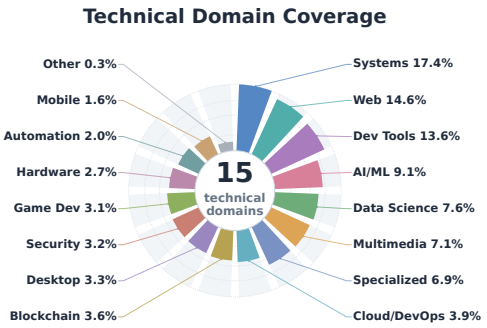

Figure 3: Technical domain coverage of the training set. The 15 labeled domains and Other (18 unlabeled tasks, 0.3%) are ordered clockwise by decreasing task share. Radial bar length is proportional to the square root of task share; percentages are computed over all 5,545 tasks in the full training dataset.

a coherent development starting point where the target functionality remains to be implemented. Throughout this process, we preserve shared components and project context with dependencies and retain the original implementation separately as a reference solution.

3.2 Execution-grounded Test Construction

We construct tests using an agent that maps the task statement's behavioral requirements to inputs and boundary cases, invokes public entry points in a reference copy of the codebase, and records the observed outcomes. Each generated test records the requirement it covers. Stateful tests also exercise specified dependencies across calls, including call ordering and cleanup.

We constrain assertions to stated requirements: reference execution establishes expected values for specified behavior, while unspecified aspects are checked only against stated constraints. We then use an agent to review every assertion for unsupported restrictions on wording, ordering, or internal structure, replacing them with behavioral checks while preserving required checks. We reject a task if an assertion depends on a private symbol and has no behavioral substitute. Revised tests are rerun on the reference solution before their inputs and assertions are fixed for grading.

3.3 Environment Preparation

Environment preparation. Starting from a uniform base container image, an agent installs dependencies and prepares build and runtime resources according to the project's declarations. Cleanup removes artifacts that could reveal the deleted implementation, including compiled outputs, cached copies, and files left by construction agents. Original tests related to the target functionality are also removed. Required packages, fixtures, and build wrappers are retained to support building and running completed implementations in the prepared environment.

Execution consistency. We check each task under the training runtime settings in six fresh containers: two with the starting codebase and four with the reference solution in place. Both starting-state runs must fail and all four reference runs must pass. These repetitions check the expected fail-to-pass transition and screen for unstable execution outcomes.

3.4 Post-rollout Filtering

Execution consistency checks evaluate the starting codebase and the reference solution. Before RL training, we further filter candidate environments using agent rollouts to probe for exploitable leakage, assess verifier decisions on generated implementations, and select tasks based on rollout outcomes.

Leakage checks. We use adversarial rollouts to test whether residual artifacts allow a solver to bypass the intended implementation work. The search covers the solver-visible environment, including compiled artifacts, caches, and installed copies. Suspected exploits are documented with supporting

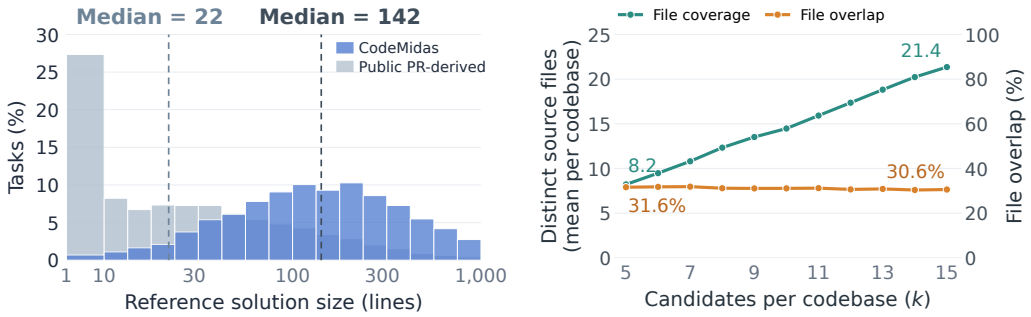

Figure 4: Implementation scope and continued exploration. Left: reference solution sizes for CodeMidas and over 30,000 tasks from public PR-derived datasets; dashed lines mark medians. Right: mean distinct source files covered (teal, left axis) and file overlap (orange, right axis) for the first k candidates in 100 selected codebases.

commands and outputs, then reviewed separately against the reference solution and verifier. We reject tasks when this review confirms that leaked material enables such a bypass.

Solution audit. We collect four agent-generated implementations per task to assess verifier decisions. We use a review agent to examine the resulting trajectories, including submitted code and test outputs, alongside the task statement, verifier, and reference solution. It assesses whether each implementation satisfies the stated requirements and compares that assessment with the test verdict. We flag a false positive when an implementation judged incorrect passes the tests, and a false negative when an implementation judged correct fails. Tasks with identified verifier defects are rejected. We further assess verifier decisions on rollouts sampled during training (Appendix E).

Rollout outcome filtering. In a separate check, we collect several solution attempts per task from a frontier model. All-pass or all-fail outcomes may reflect task difficulty or residual defects, such as weak tests or requirements missing from the statement. We retain only tasks with both successful and failed attempts under the screening model and rollout budget.

3.5 Task Supply and Diversity

Applying the construction and filtering pipeline yields 5,545 tasks from 3,185 codebases across 23 programming languages and 15 technical domains (Figures 2 and 3). Their median reference solution size is 142 lines, compared with 22 for over 30,000 tasks from public PR-derived datasets (Figure 4, left; Appendix B.1). We examine task boundaries relative to development artifacts, continued exploration within codebases, and functional diversity across codebases.

Tasks from implemented functionality. To examine whether these tasks are explicitly defined in development artifacts, we randomly sample 164 tasks and review their associated PRs, issues, and commit histories. We check whether PRs and issues contain the complete task requirements, and whether a commit or group of commits clearly defines the task. We find that **59.8%** of tasks span multiple commits; for another **20.1%**, we identify no commit or combination of commits that clearly defines the complete task. We find no complete requirement coverage for **77.4%** of tasks in the reviewed PRs and **99.4%** in issues (Appendix B.2). CodeMidas can thus turn implemented functionality into tasks even when development artifacts do not provide complete task definitions.

Exploration and task diversity. To examine whether expanding task supply leads to repetition, we study diversity within and across codebases. We first compare file overlap among tasks from the same codebase. The mean overlap is lower in our training set than in the public PR-derived pool (29.3% versus 39.8%; Appendix B.1). We then test whether exploration remains productive beyond the original limit of five candidates per codebase by extending task design to 15 candidates in 100 selected codebases. From 5 to 15 candidates, the mean number of covered source files grows by a factor of 2.6, while overlap remains near 31% (Figure 4, right; Appendix B.3). We further examine whether broader codebase coverage also brings distinct functional requirements within the same domain. Among the 15 labeled domains, we select those with the highest, median, and lowest task shares to cover different levels of representation in the training set. Within each domain, we randomly sample 30 codebases and then randomly select one task from each. Three rounds of model screening and human review of flagged pairs identify no near-duplicates (Appendix B.4). These

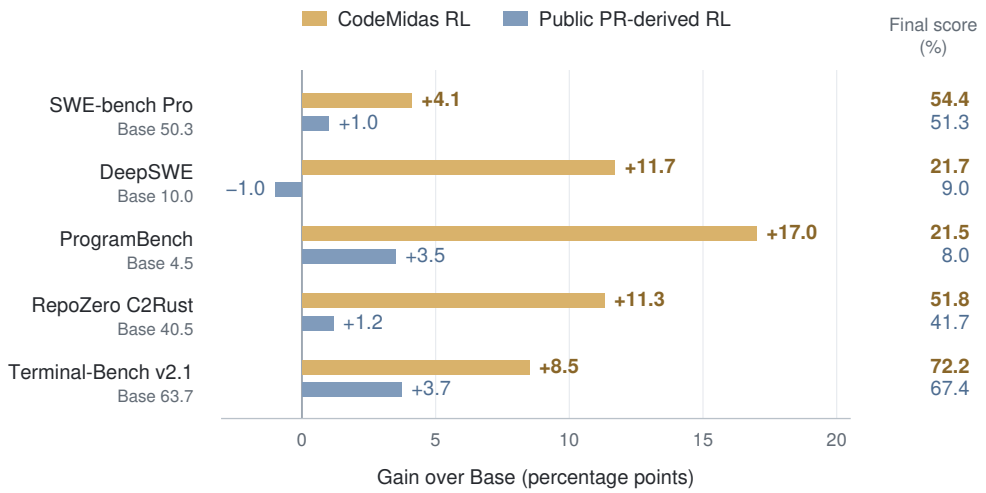

Figure 5: Performance gains from RL on CodeMidas and public PR-derived datasets. Gold and blue bars show absolute improvements over the common initial policy (Base), in percentage points. Initial scores appear below the benchmark names, and final scores appear on the right.

results support continued exploration within codebases and functional diversity across codebases, providing complementary evidence for expanding the supply of coding RL tasks.

4 Experiments

We evaluate the training value of environments produced by CodeMidas's full pipeline (§3), comparing against public PR-derived training datasets (Zhao et al., 2026; Badertdinov et al., 2026; Tao et al., 2026; Zan et al., 2025), pooled as described in Appendix B.1, under shared RL settings (§4.1; Appendix C) and assessing reward reliability during training (§4.2; Appendix E).

4.1 Experimental Setup

We train MiMo-V2.5 (Xiaomi MiMo Team, 2026b) on 5,545 CodeMidas tasks with GRPO (Shao et al., 2024), binary execution rewards, replacing groups whose rollouts receive identical rewards with groups from newly sampled tasks (Yu et al., 2026). See Appendix C. For comparison, we train a Public PR-derived RL baseline on the same pool of over 30,000 tasks from public PR-derived datasets used in Section 3.5 (Appendix B.1), using the same setting.

We use identical evaluation settings for the initial policy, CodeMidas RL, and Public PR-derived RL checkpoints, with three attempts per task on every benchmark. We follow the official task sets for the five external benchmarks: SWE-bench Pro (Deng et al., 2026), DeepSWE v1.1 (Huang & Jiang, 2026), ProgramBench (Yang et al., 2026), RepoZero C2Rust (Zhang et al., 2026b), and Terminal-Bench v2.1 (Merrill et al., 2026). CodeMidas Val consists of 200 randomly sampled CodeMidas tasks separate from the 5,545 training tasks. We verified that the CodeMidas training set shares no source codebases with all six benchmark task sets. For each run, we report all benchmark results at the checkpoint with the highest pass rate on CodeMidas Val. For ProgramBench, we report Almost Resolved, the percentage of tasks passing at least 95% of their tests; all other evaluations report pass rate. For each benchmark, we report absolute score improvements relative to the initial policy.

4.2 Results

RL improves performance across task types. Training on CodeMidas improves performance on all five external benchmarks (Figure 5). DeepSWE pass rate increases from 10.0% to 21.7%, while Terminal-Bench v2.1 improves from 63.7% to 72.2%. On ProgramBench, the Almost Resolved score rises from 4.5 to 21.5. The gains span codebase repair, code translation, program construction, and terminal work, supporting the use of source-derived tasks to train agents for diverse software work.

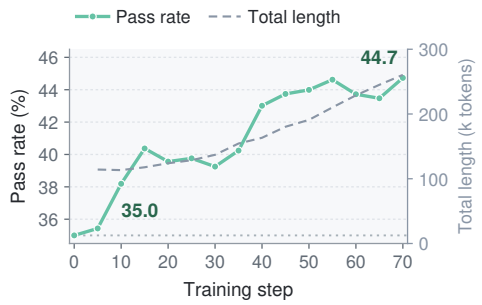

Figure 6: RL dynamics on the full pool. Pass rate (green, left axis) and mean total length (gray-blue dashed, right axis) during RL on CodeMidas. Length is measured in thousands of tokens; the horizontal dotted line marks the pass rate achieved by the initial policy.

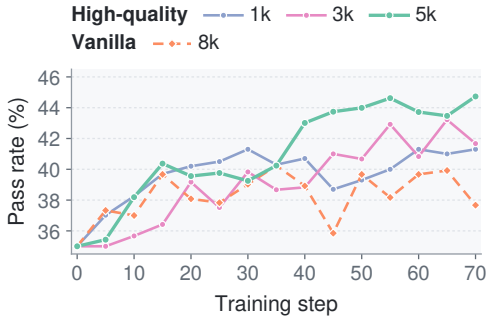

Figure 7: Task-pool comparison on CodeMidas Val. Pass rates at five-step intervals. Solid lines with circles denote the high-quality 1k, 3k, and full (5k) pools; the dashed line with diamonds denotes the vanilla 8k sample drawn before filtering and cleaning.

CodeMidas provides stronger gains than public PR-derived datasets. Under the same training and evaluation settings, RL on 5,545 CodeMidas tasks outperforms RL on over 30,000 tasks from public PR-derived datasets across all five external benchmarks (Figure 5). The advantage ranges from 3.1 percentage points on SWE-bench Pro to 13.5 on ProgramBench. These results support the training value of CodeMidas's complete construction and filtering pipeline.

Benefits extend to a smaller model. Training MiMo-V2.6-Distill-Qwen-9B (Xiaomi MiMo Team, 2026a) on the same 5,545 tasks increases its pass rate from 37.1% to 42.0% on Terminal-Bench v2.1 and from 31.0% to 38.1% on RepoZero C2Rust. These results support the training value of CodeMidas for a smaller initial policy. Appendix D reports the training settings and additional results.

Learning dynamics on CodeMidas Val. Figure 6 tracks pass rate and mean total token length during RL on CodeMidas. Pass rate rises from 35.0% to 44.7%. From step 40 onward, evaluated checkpoints remain roughly 8–10 percentage points above the initial rate. These gains accompany longer trajectories, indicating greater use of the available interaction budget.

Reward reliability during training. We audit 1,000 rollout groups from training steps 1–70 (500 accepted for training and 500 discarded) for known exploit patterns and find no confirmed instances of the targeted submission-level patterns in the reviewed cases. Confirmed reference-code leakage occurs in 0.1% of verifier-accepted submissions within the sampled groups used for training. A human review assesses sampled submissions against task requirements. Among accepted submissions, main-usage defects are observed in 6.2% of early and 4.4% of later samples. These findings provide complementary evidence for reward reliability during RL (Appendix E).

5 Analysis

To understand the gains from training on CodeMidas, we examine the contributions of task scale and quality (Section 5.1). We then analyze how agent behavior changes during RL, how these behaviors are associated with task success, and whether they generalize across task types (Section 5.2).

5.1 Task Scale and Quality

To examine the effects of task scale and quality, we train on the full high-quality CodeMidas dataset of 5,545 tasks (5k) and random 1k and 3k subsets. We also train on approximately 8,000 tasks (8k) sampled before filtering, each with a task statement, development environment, and verifier. This vanilla 8k sample is used without environment cleaning or execution consistency checks (Section 3.3), or any of the three post-rollout filtering steps described in Section 3.4.

All four settings use identical training configurations and are evaluated over the same checkpoint range. We first examine learning curves on CodeMidas Val, then compare scores on SWE-bench Pro, DeepSWE, and CodeMidas Val for each of the four training settings.

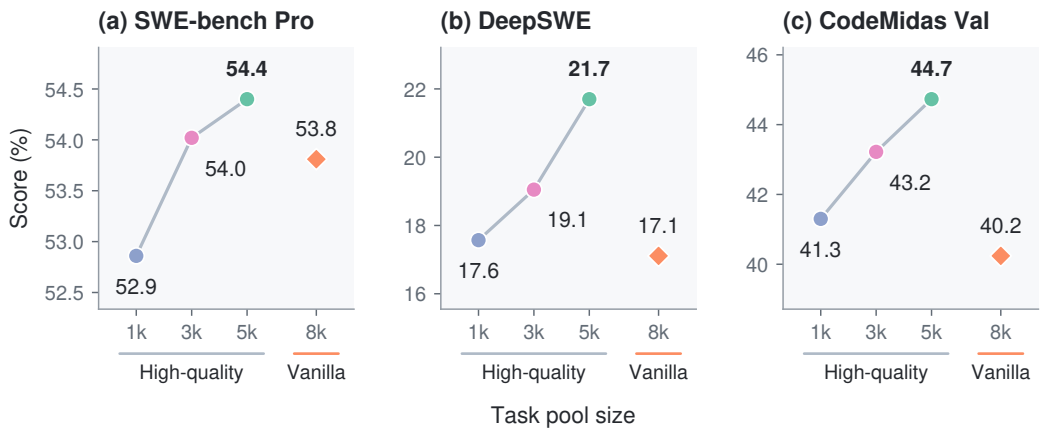

Figure 8: Task scale and quality. Scores on SWE-bench Pro, DeepSWE, and CodeMidas Val. Connected circles denote high-quality 1k, 3k, and 5k pools; diamonds denote vanilla 8k. Panels use different y-axis ranges.

The full dataset leads across later checkpoints. We first examine every evaluated checkpoint on CodeMidas Val (Figure 7). The 1k setting reaches 41.3 at step 30, while the 3k setting reaches 43.2 at step 65. The full dataset leads at every evaluated checkpoint from step 40 through step 70, where it reaches 44.7. Its advantage thus persists across the later part of training.

Performance improves with task scale. We next compare scores across all three evaluations (Figure 8). The 1k, 3k, and 5k pools achieve progressively higher scores: DeepSWE scores rise from 17.6 to 19.1 to 21.7, while scores on CodeMidas Val increase from 41.3 to 43.2 to 44.7. SWE-bench Pro follows the same trend with a smaller additional gain from 3k to 5k. These results support scaling high-quality training data, with gains extending to external benchmarks.

The high-quality 5k pool outperforms the vanilla 8k sample. The full CodeMidas dataset exceeds the vanilla 8k sample by 0.6, 4.6, and 4.5 percentage points on SWE-bench Pro, DeepSWE, and CodeMidas Val, respectively. Even the high-quality 3k subset outperforms the vanilla 8k sample on all three evaluations. These comparisons support the combined benefits of environment cleanup, execution consistency checks, and post-rollout filtering for RL training (Appendix A).

5.2 Behavioral Changes and Generalization

To characterize the behavioral changes accompanying the performance gains, we analyze codebase exploration, reasoning before code edits, and self-verification (Figure 9). For reasoning before edits, we measure the fraction of written code fragments that appeared in preceding reasoning.

Exploration and drafting increase; self-verification diversifies. To quantify behavioral changes during RL, we compare early and late training rollouts (Table 2). We measure exploration by read/search calls before the first edit, drafting by the fraction of code fragments in `Write`/`Edit` payloads already present in preceding reasoning, and self-verification by distinct verification commands after the final codebase edit. Exploration increases from 27.2 to 40.1 calls, the drafting ratio from 35.8% to 62.9%, and verification commands from 2.0 to 2.5. Agents thus explore more, show greater overlap between written code and preceding reasoning, and execute more diverse verification commands. Metric definitions and aggregation procedures are given in Appendix F.

Self-verification is associated with higher pass rates. On CodeMidas Val, we compare the pass rates of rollouts with and without checks that the agent itself writes and executes, holding the task and checkpoint fixed in each comparison. Averaging the differences equally across tasks gives +4.2 percentage points. The same aggregation for checkpoint-median splits of exploration and drafting, excluding undefined measurements, gives +0.7 and +2.0 points, respectively.

Behavioral changes generalize; interaction length varies by task. To assess generalization beyond CodeMidas, we compare early and late checkpoints on SWE-bench Pro, ProgramBench, and Terminal-Bench v2.1 (Table 3). We also measure interaction length in assistant turns to examine how it changes across task types. Exploration increases across all three benchmarks, while changes in verification

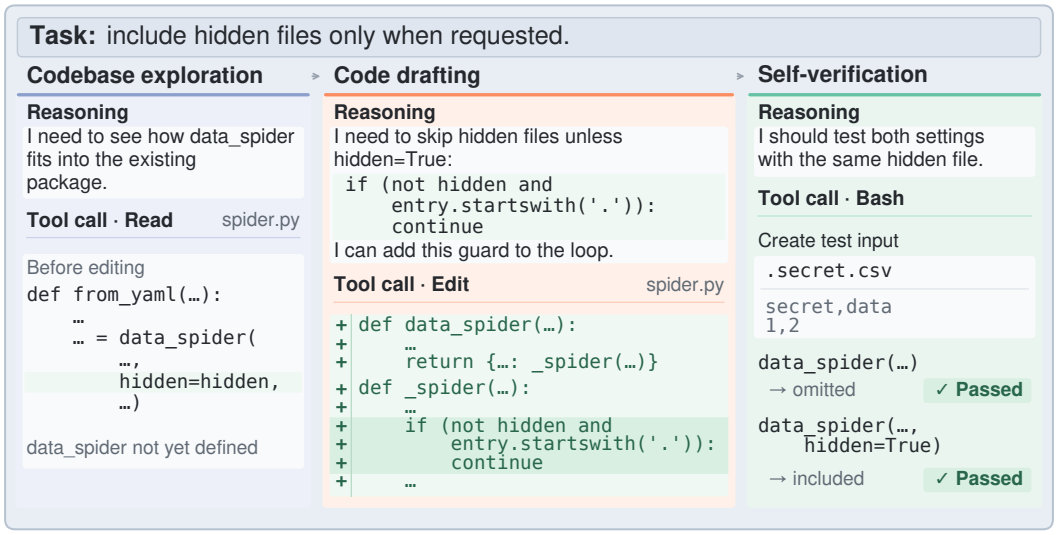

Figure 9: One rollout exhibits the three behaviors measured in Table 2. Columns pair reasoning with tool use: reading the caller, drafting hidden-file filtering and applying it via `Edit`, and testing both flag settings with `.secret.csv`. The diff marks additions; `Passed` indicates agreement with expected behavior.

diversity vary by benchmark. Drafting ratios increase on SWE-bench Pro and ProgramBench. Mean interaction length increases from 37.3 to 50.1 turns for issue repair on SWE-bench Pro and decreases from 155.1 to 122.8 for whole-program construction on ProgramBench. On ProgramBench, greater codebase exploration thus accompanies shorter overall interactions.

Table 2: Pre-edit exploration and code drafting rise sharply during RL; verification diversity rises slightly.

Behavior	Measure	Early	Late	Change
Codebase exploration	Pre-edit read/search calls	27.2	40.1	+12.9
Code drafting	Drafting ratio (%)	35.8	62.9	+27.1
Self-verification	Distinct post-edit commands	2.0	2.5	+0.5

Table 3: Behavior on held-out tasks. Means over the first → last three observed checkpoints for each benchmark. The three behavioral measures follow Table 2 and include rollouts with defined measurements; interaction length includes all scored rollouts. Measured subsets can vary across checkpoints.

Evaluation	Codebase exploration (read/search calls)	Code drafting (drafting ratio, %)	Self-verification (distinct commands)	Interaction length (assistant turns)
SWE-bench Pro	23.1 → 35.5	30.4 → 65.3	0.8 → 1.0	37.3 → 50.1
ProgramBench	55.7 → 83.6	10.6 → 36.1	0.9 → 1.0	155.1 → 122.8
Terminal-Bench v2.1	11.9 → 16.8	N/A	2.0 → 2.6	59.2 → 69.5

6 Conclusion

CodeMidas demonstrates the value of existing codebases as a source of coding RL data. Training on 5,545 tasks improves MiMo-V2.5 across five external benchmarks, with gains spanning diverse forms of software work. Analyses show that expanding high-quality task pools improves performance and that a smaller filtered dataset can outperform a larger unfiltered one. During training, agents explore codebases more and perform more varied self-verification; agent-written checks are associated with higher success rates, and these changes also appear on external tasks.

CodeMidas applies a Midas touch to existing codebases, turning implemented functionality into effective training environments and providing a practical route to scaling coding RL from code itself.

AI USAGE DISCLOSURE

We used LLM agents to construct and validate coding tasks and environments, with human review of sampled outputs. We also used LLMs for manuscript feedback and language refinement. The authors take full responsibility for the final content of this work.

REFERENCES

Anthropic. Claude Code, 2025. URL https://github.com/anthropics/claude-code.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench V2: Language-Agnostic SWE Task Collection at Scale. In *Proceedings of the 43rd International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=UCAda9kS57.

Bowen Baker, Joost Huizinga, Leo Gao, Zehao Dou, Melody Y. Guan, Aleksander Madry, Wojciech Zaremba, Jakub Pachocki, and David Farhi. Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation. arXiv preprint arXiv:2503.11926, 2025. URL https://arxiv.org/abs/2503.11926.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015. doi: 10.1109/TSE.2014.2372785. URL https://doi.org/10.1109/TSE.2014.2372785.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. CodeT: Code Generation with Generated Tests. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, Hao Ge, Zongmeng Zhang, Zeyu Cui, Dayiheng Liu, Jingren Zhou, Jianling Sun, Junyang Lin, and Binyuan Hui. SWE-Universe: Scale Real-World Verifiable Environments to Millions. arXiv preprint arXiv:2602.02361, 2026a. URL https://arxiv.org/abs/2602.02361.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching Large Language Models to Self-Debug. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=KuPixIqPiq.

Yihao Chen, Shi Chang, Khaled Chawa, Feng Lin, Boyuan Chen, Shaowei Wang, and Ahmed E. Hassan. MindForge: Teaching Small Language Models Whole-Life-Cycle Software Engineering via Source-Free Program Synthesis. arXiv preprint arXiv:2607.27146, 2026b. URL https://arxiv.org/abs/2607.27146.

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://openreview.net/forum?id=uEVTdoAbnK.

Mahir Labib Dihan and Md Ashrafur Rahman Khan. SWE-Shepherd: Advancing PRMs for Reinforcing Code Agents. arXiv preprint arXiv:2604.10493, 2026. URL https://arxiv.org/abs/2604.10493.

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 End-to-End Feature-Driven Software Development. arXiv preprint arXiv:2505.16975, 2025. URL https://arxiv.org/abs/2505.16975.

Dayuan Fu, Shenyu Wu, Yunze Wu, Zerui Peng, Yaxing Huang, Jie Sun, Ji Zeng, Mohan Jiang, Lin Zhang, Yukun Li, Jiarui Hu, Liming Liu, Jinlong Hou, and Pengfei Liu. daVinci-Env: Open SWE Environment Synthesis at Scale. arXiv preprint arXiv:2603.13023, 2026. URL https://arxiv.org/abs/2603.13023.

Zhongmou He, Yee Man Choi, Kexun Zhang, Jiabao Ji, Junting Zhou, Dejia Xu, Ivan Bercovich, Aidan Zhang, and Lei Li. HardTests: Synthesizing High-Quality Test Cases for LLM Coding. arXiv preprint arXiv:2505.24098, 2025. URL https://arxiv.org/abs/2505.24098.

Wenqi Huang and Peter Jiang. DeepSWE v1.1, 2026. URL https://deepswe.datacurve.ai/blog/deepswe-v1-1.

Naman Jain, Manish Shetty, Tianjun Zhang, King Han, Koushik Sen, and Ion Stoica. R2E: Turning any Github Repository into a Programming Agent Environment. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 21196–21224. PMLR, 2024. URL https://proceedings.mlr.press/v235/jain24c.html.

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.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Minseon Kim, Zhengyan Shi, Emiliano Penaloza, Christopher Cui, Roger Creus Castanyer, Maryam Hashemzadeh, Isadora White, Jonathan Light, Jeonghye Kim, Matheus Pereira, Darya Moldavskaya, Chinmay Singh, Fabio Vera, Baolin Peng, Xingdi Yuan, Marc-Alexandre Côté, and Alessandro Sordoni. FrogNano: Training a 4B Coding Agent via Online Task Synthesis. arXiv preprint arXiv:2609.07925, 2026. URL https://arxiv.org/abs/2609.07925.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In *Advances in Neural Information Processing Systems*, volume 35, pp. 21314–21328. Curran Associates, Inc., 2022. doi: 10.52202/068431-1549. URL https://doi.org/10.52202/068431-1549.

Jiarong Liang, Zhiheng Lyu, Zijie Liu, Xiangchao Chen, Ping Nie, Kai Zou, and Wenhu Chen. SWE-Next: Scalable Real-World Software Engineering Tasks for Agents. arXiv preprint arXiv:2603.20691, 2026. URL https://arxiv.org/abs/2603.20691.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation. In *Advances in Neural Information Processing Systems*, volume 36, pp. 21558–21572. Curran Associates, Inc., 2023. doi: 10.52202/075280-0943. URL https://doi.org/10.52202/075280-0943.

Zichen Liu, Changyu Chen, Wenjun Li, Penghui Qi, Tianyu Pang, Chao Du, Wee Sun Lee, and Min Lin. Understanding R1-Zero-Like Training: A Critical Perspective. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=5PAF7PAY2Y.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sebastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. StarCoder 2 and The Stack v2: The Next Generation. arXiv preprint arXiv:2402.19173, 2024. URL https://arxiv.org/abs/2402.19173.

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Tarun Venkat, Shang Zhu, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL, 2025. URL https://www.together.ai/blog/deepswe.

Mike A Merrill, Alexander Glenn Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon 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 Kumar Guha, Gabriel H. S. Dreiman, Jiacheng Zhu, Karl Krauth, Li Zhong, Niklas Muennighoff, Robert Kwesi Amanfu, Shangyin Tan, Shreyas Pimpalgaonkar, Tushar Aggarwal, Xiangning Lin, Xin Lan, Xuandong Zhao, Yiqing Liang, Yuanli Wang, Zilong 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 Michael 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 *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=a7Qa4CcHak.

OpenAI. Introducing SWE-bench Verified, 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Mohit Raghavendra, Anisha Gunjal, Bing Liu, and Yunzhong He. Agentic Rubrics as Contextual Verifiers for SWE Agents. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 15265–15290. Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.acl-long.697. URL https://aclanthology.org/2026.acl-long.697/.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models. arXiv preprint arXiv:2402.03300, 2024. URL https://arxiv.org/abs/2402.03300.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 36, pp. 8634–8652. Curran Associates, Inc., 2023. doi: 10.52202/075280-0377. URL https://doi.org/10.52202/075280-0377.

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.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *The Thirteenth International Conference on Learning Representations*, 2025a. URL https://openreview.net/forum?id=OJd3ayDDoF.

You Wang, Michael Pradel, and Zhongxin Liu. Are "Solved Issues" in SWE-bench Really Solved Correctly? An Empirical Study. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pp. 169–181. Association for Computing Machinery, 2026. doi: 10.1145/3744916.3764576. URL https://doi.org/10.1145/3744916.3764576.

Zihan Wang, Siyao Liu, Yang Sun, Ming Ding, and Hongyan Li. CodeContests+: High-quality test case generation for competitive programming. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5576–5600, Suzhou, China, 2025b. Association for Computational Linguistics. ISBN 979-8-89176-335-7. doi: 10.18653/v1/2025.findings-emnlp.299. URL https://aclanthology.org/2025.findings-emnlp.299/.

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution. In *Advances in Neural Information Processing Systems*, volume 38, pp. 78500–78525. Curran Associates, Inc., 2025. doi: 10.52202/085713-2629. URL https://doi.org/10.52202/085713-2629.

Yuxiang Wei, Zhiqing Sun, Emily McMilin, Jonas Gehring, David W. Zhang, Gabriel Synnaeve, Daniel Fried, Lingming Zhang, and Sida Wang. Toward Training Superintelligent Software Agents through Self-Play SWE-RL. In *Forty-third International Conference on Machine Learning*, 2026. URL https://openreview.net/forum?id=0ophJB76sC.

Chuan Xiao, Zhengbo Jiao, Shaobo Wang, Wei Wang, Bing Zhao, Hu Wei, Linfeng Zhang, and Lin Qu. Socratic-SWE: Self-Evolving Coding Agents via Trace-Derived Agent Skills. arXiv preprint arXiv:2606.07412, 2026. URL https://arxiv.org/abs/2606.07412.

Xiaomi MiMo Team. MiMo-V2.6: Scaling Reinforcement Learning Towards Self-Improvement. https://huggingface.co/XiaomiMiMo/MiMo-V2.6-Pro-RL, 2026a.

Xiaomi MiMo Team. MiMo-V2.5, 2026b. URL https://mimo.xiaomi.com/mimo-v2-5.

Yiqing Xie, Alex Xie, Divyanshu Sheth, Pengfei Liu, Daniel Fried, and Carolyn Rose. RepoST: Scalable Repository-Level Coding Environment Construction with Sandbox Testing. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=2txrMBpw3q.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://doi.org/10.52202/079017-1601.

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. Curran Associates, Inc., 2025. doi: 10.52202/085713-3239. URL https://doi.org/10.52202/085713-3239.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can Language Models Rebuild Programs From Scratch? arXiv preprint arXiv:2605.03546, 2026. URL https://arxiv.org/abs/2605.03546.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, et al. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244, 2026.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Aoyan Li, Lu Chen, Xiaojian Zhong, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Ming Ding, and Liang Xiang. Multi-SWE-bench: A multilingual benchmark for issue resolving. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL https://papers.nips.cc/paper_files/paper/2025/hash/5afa9cb1e917b898ad418216dc726fbd-Abstract-Datasets_and_Benchmarks_Track.html.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. ACECODER: Acing Coder RL via Automated Test-Case Synthesis. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12023–12040. Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.acl-long.587. URL https://aclanthology.org/2025.acl-long.587/.

Yucheng Zeng, Shupeng Li, Daxiang Dong, Ruijie Xu, Zimo Chen, Liwei Zheng, Yuxuan Li, Zhe Zhou, Haotian Zhao, Lun Tian, Heng Xiao, Tianshu Zhu, Longkun Hao, and Jianmin Wu. SWE-Hub: A Unified Production System for Scalable, Executable Software Engineering Tasks. arXiv preprint arXiv:2603.00575, 2026. URL https://arxiv.org/abs/2603.00575.

Lei Zhang, Jiaxi Yang, Min Yang, Jian Yang, Mouxiang Chen, Jiajun Zhang, Zeyu Cui, Binyuan Hui, and Junyang Lin. Synthesizing Software Engineering Data in a Test-Driven Manner. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 76518–76540. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhang25cn.html.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. Swe-bench goes live! *Advances in Neural Information Processing Systems*, 38, 2026a.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. RepoZero: Can LLMs Generate a Code Repository from

Scratch? arXiv preprint arXiv:2605.07122, 2026b. URL https://arxiv.org/abs/2605.07122.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne 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, 2026. URL https://arxiv.org/abs/2602.09892.

A Construction and Filtering Diagnostics

This appendix examines how the six stages in Figure 1 turn source-derived candidates into RL environments. The goals are coherent tasks with executable tests, rewards aligned with the stated requirements, and task selection for quality and training efficiency. Table A1 reports attrition across the full pipeline; the selected cases illustrate distinct failure mechanisms rather than their prevalence.

Reference success does not establish reward reliability. Among the 11,930 candidates retained after execution consistency and leakage checks, the reviewing agent identifies verifier defects in **3,757 (31.5%)**. Of these rejected tasks, **81.5%** involve false negatives: 68.5% have false negatives only and 13.0% have both error types. These findings motivate checking verifier agreement with the requirements across alternative implementations, beyond successful execution of the reference solution.

Table A1: Attrition across the construction and filtering pipeline. Retained counts follow Figure 1. Removal rates use incoming tasks as their denominator; reason percentages use tasks not retained at that stage. Stages are sequential, so each distribution is conditional on preceding stages.

Stage	Retained	Removed (rate)	Reasons among tasks not retained
Task design	22,575	—	Agent rejection: 91.7% Infrastructure failure: 8.3%
Test construction	16,027	6,548 (29.0%)	Agent rejection: 54.0% Infrastructure failure: 46.0%
Execution consistency	12,746	3,281 (20.5%)	Reference solution fails: 84.7% Starting codebase passes: 4.1% Infrastructure failure: 11.3%
Leakage checks	11,930	816 (6.4%)	Flagged as exploitable: 98.8% Infrastructure failure: 1.2%
Solution audit	8,173	3,757 (31.5%)	False negative only: 68.5% False positive only: 18.5% Both: 13.0%
Rollout outcome filtering	5,545	2,628 (32.2%)	All attempts fail: 76.0% All attempts pass: 24.0%

Task design is the first retained count; its input count and removal rate are not reported. Infrastructure failures are distinguished from task rejections. False-positive and false-negative categories follow the reviewing agent's assessment against task requirements. Rollout outcomes refer to screening attempts. Percentages may not sum to 100% due to rounding.

A.1 Task Design

Task design must isolate meaningful implementation work while preserving a usable project. In a database query compiler, the C++ code generator shares its core logic with another backend. Removing only the generator's wrapper leaves most target functionality implemented, while removing the shared logic also affects the other backend. Neither boundary yields the intended task. This case illustrates why task boundaries must account for dependencies in the surrounding codebase.

A.2 Test Construction

Test construction must establish agreement between the task statement and reference behavior before fixing test expectations. In a JSON-repair library, the reference implementation fails an example included in the statement itself. The statement and reference execution therefore specify conflicting results for the same input. This candidate cannot provide a consistent behavioral contract for the solver and verifier, motivating joint validation of the requirements and tests.

A.3 Execution Consistency

In six fresh containers, both starting-state runs must fail and all four reference-solution runs must pass. A browser scroll-animation polyfill illustrates the importance of checking the starting state: all nine tests pass even after the target functionality has been deleted. The verifier thus awards success without requiring the missing implementation. The execution checks assess this distinction between known code states; subsequent rollouts examine shortcuts and alternative implementations.

A.4 Post-rollout Filtering: Leakage Checks

Leakage checks examine whether the solver-visible environment exposes a route to reward that bypasses the intended implementation work. In a Java project, the Maven cache contains a source archive whose relevant code matches the reference solution line for line. A solver can recover the implementation by reading the archive despite its removal from the working source tree. This motivates probing the broader environment, including dependencies and caches, after source cleanup.

A.5 Post-rollout Filtering: Solution Audit

The audit evaluates verifier decisions on agent-generated implementations against the stated requirements (Section 3.4). The two cases below illustrate both directions of error that can remain after the preceding checks.

False negative: rejecting correct behavior. A theorem prover's test enforces an accidental double space in the reference output. All four agent attempts are judged correct but receive zero reward because of this comparison. An unspecified formatting choice has become a requirement for acceptance.

False positive: rewarding an incorrect implementation. In an audio-analysis tool, the test script prints assertion failures but exits with code `0`. A submitted stub that produces no output consequently receives full reward. The successful exit status prevents the failed checks from reaching the reward signal.

A.6 Post-rollout Filtering: Rollout Outcomes

Rollout outcome filtering targets **task quality and training efficiency**. All-pass or all-fail outcomes can reflect residual defects, such as weak tests or missing requirements, as well as task difficulty under the screening model. We retain only tasks with both successful and failed attempts.

Our GRPO implementation already discards and replaces groups with identical rewards. Generating and verifying their rollouts still incurs computation, so screening tasks before RL aims to reduce the repeated collection of groups that provide no training update. This further quality screen complements the preceding audits. Its outcomes depend on the screening model and budget and need not persist under the training policy: 31.8% of sampled groups in the 5k run still receive identical rewards (Appendix C).

Connection to Training Results

Section 5.1 evaluates the combined training value of the complete procedure, including the task distribution induced by outcome filtering. These diagnostics explain the roles of individual checks without isolating their downstream gains. Appendix E separately examines residual reward defects during RL.

B Task Supply and Diversity

We examine how source-based construction affects the scope and supply of coding RL tasks. We first define the public PR-derived comparison pool and compare implementation scope and file overlap. We then examine task boundaries in development history, continued exploration within codebases, and functional diversity across codebases. These analyses complement the training results in Sections 4 and 5.1.

B.1 Comparison with Public PR-derived Tasks

Public PR-derived pool. The baseline contains 30,128 tasks: Scale-SWE (Zhao et al., 2026) (17,202), SWE-rebench V2 (Badertdinov et al., 2026) (6,262), the real-data subset of SWE-Lego (Tao et al., 2026) (4,432), and Multi-SWE-RL (Zan et al., 2025) (2,232). PRs supply reference patches; linked issues supply statements except in Scale-SWE, which synthesizes them from PRs. We use released tasks after confirming that each reference patch passes its tests in our environment. The comparison evaluates CodeMidas's full pipeline against existing public training resources, so this pool receives no CodeMidas post-rollout filtering; both RL runs discard and replace groups with identical rewards (Table C6). The same pool supports the dataset comparisons in Section 3.5 and the RL baseline in Section 4.1.

Broader reference implementations with lower file overlap. Reference size counts added or deleted source lines, including comments and blank lines. CodeMidas has a median of 142 lines (IQR 66–305), versus 22 for the public pool; 65.9% versus 45.3% of patches span multiple source files. For each task, file overlap is the fraction of other tasks in its codebase sharing a changed source file. Averaging within codebases and then equally across eligible codebases gives 29.3% versus 39.8%. Figure 10 compares the full task pools by reference solution size. These measurements describe implementation scope and shared file locations; the following analyses examine task boundaries and opportunities for expanding the supply.

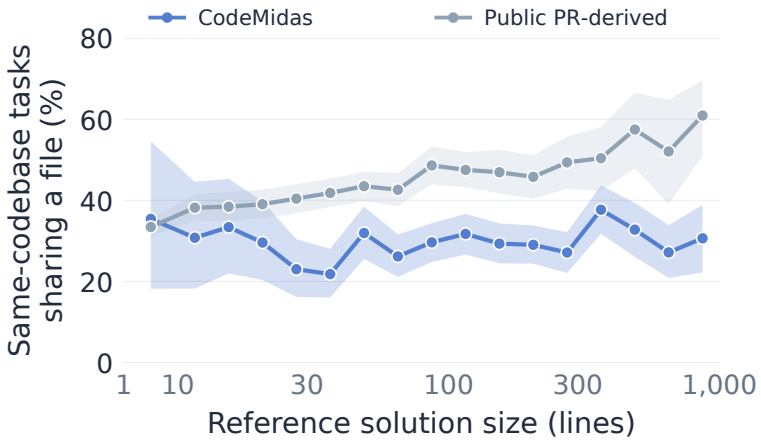

Figure 10: File overlap of the CodeMidas training set and public PR-derived tasks. The full task pools are compared by reference solution size. Task-level overlap is weighted within each bin by the inverse number of tasks in its codebase. Shading shows 95% confidence intervals from resampling codebases.

Size bins and weighting. Reference-size plots use the range 1–1,000 lines, with the 1–10-line bin occupying one fifth the width of each subsequent logarithmic decade. Reference-size percentages in Figure 4 (left) use each full task pool as the denominator. Overlap curves weight tasks within each bin by the inverse number of eligible tasks in their codebase. Confidence intervals in Figure 10 resample codebases.

B.2 Task Boundaries beyond Development Artifacts

A functional task can span several changes or occupy part of a broad import. In an audit of 164 randomly sampled tasks from 100 codebases, only 33 tasks (20.1%) map to a single commit that clearly defines the complete task. Another 98 (59.8%) span multiple commits; for 33 (20.1%), no clear commit grouping is identified. Complete requirements are found in the reviewed PRs for 37 tasks and in issues for one task. Thus, the audit identifies no complete requirement coverage for 77.4% and 99.4% of tasks, respectively (Table B2). CodeMidas can define tasks around coherent functionality even when the reviewed development artifacts do not supply a matching task boundary or complete specification.

Table B2: Task boundaries and requirements in development artifacts. Audit of 164 tasks from 100 codebases. Commit categories concern a clearly defined task boundary; PR/issue categories concern requirement coverage.

Commit grouping	**Tasks**	**Share**
One commit	33	20.1%
Multiple commits	98	59.8%
No clear commit grouping identified	33	20.1%
PR / issue coverage	**PR**	**Issue**
Complete coverage in one record	25	0
Complete coverage across records	12	1
Partial coverage	58	93
No matching record identified	69	70

How the boundaries differ. The `tributary` task combines a file-source adapter and its export (`5adaedc`) with later CSV support, revised defaults, and option checks (`811a7e1`). The `calculatex` task combines raw LaTeX handling (`dba4ab53`) with error handling for unclosed input enabled by later infrastructure (`a16ebd5`). Conversely, the `Grumpy` task isolates generator creation, yield suspension, and `send`/resume behavior from an initial import of an entire compiler, runtime, libraries, and build system. The reviewed PRs and issues also lack its complete generator requirements. Source code exposes these functional units even when historical changes have different boundaries.

Audit protocol and scope. The AI-assisted review retains record links and per-task rationales, revisits historical diffs to resolve disputed judgments, and uses a 15-minute budget per task in the final supplementary review. PR/issue categories distinguish complete coverage in one or multiple records, partial coverage, and no matching record identified. The last category includes 64 no-match cases and five insufficient-evidence cases for PRs, and 70 no-match cases for issues. The reported counts include 13 PR and 13 issue reviews without a valid decision within budget. They measure coverage identified by this audit, rather than proving absence or measuring existing pipelines' extraction failure rates.

The 98 multiple-commit tasks comprise 85 cases of successive evolution of the same functionality and 13 combining different functional changes; this distinction depends on functionality granularity. The 33 tasks without a clear grouping comprise 25 broad imports or changes without a clear task boundary, six insufficient-evidence cases, and two without a delivered decision. The audit concerns how records delineate a task, rather than whether its implementation exists in commit history.

B.3 Continued Exploration Expands Source Coverage

Coverage grows 2.6-fold while file overlap remains near 31%. Extending task design from five to 15 candidates in 100 selected codebases increases mean distinct source-file coverage from 8.21 to 21.35 (Figure 4, right). Each additional candidate position contributes an average of 0.95–1.53 previously uncovered files per codebase, while mean overlap stays between 30.34% and 31.84%. These results show continued exploration of new implementation regions at a similar level of file overlap. The additional candidates have not passed the full RL-environment filtering pipeline.

Table B3: Continued exploration reaches additional source files. Coverage averages over all 100 codebases; overlap uses codebases with at least two nonempty source patches.

k	Observed source patches	Mean distinct source files	Mean overlap (%)	Codebases with calculable overlap
3	232	5.6	40.1	84
5	**387**	**8.2**	**31.6**	**90**
6	461	9.5	31.8	93
9	673	13.5	31.1	94
12	884	17.4	30.6	95
15	**1,096**	**21.4**	**30.6**	**96**

Generation and measurement. We analyze the first k candidate positions in generation order for $k = 3, \ldots, 15$. The 1,500 positions yield 1,097 statements with patches, 331 rejections, 30 construction errors, and 42 missing outcomes; 1,096 patches contain source-file changes. Each codebase contributes one session, with five scheduled replacement sessions used in full. Within-session retries replace only incomplete or failed patch construction, retaining completed successes and rejections. Coverage averages the union of changed source files over all 100 codebases; rejected, failed, and missing candidates add no observed files. Overlap averages the fraction of unordered nonempty-patch pairs sharing a file, equally across eligible codebases (Table B3).

Comparison with public PR-derived tasks. We subsample the pool defined in Appendix B.1 using a fixed random order (seed 20260923), taking up to k available tasks per codebase. All 4,233 normalized codebases remain in the cohort; 2,536 have at least two nonempty patches and hence defined overlap at every displayed k. CodeMidas shows lower overlap in most reference-size bins at each $k = 6, 9, 12, 15$ (Figure 11). This pattern persists as more candidates are included. The curves use the size bins and weighting in Appendix B.1 and include small bins; different codebase populations and available task counts make this a descriptive comparison of shared file locations.

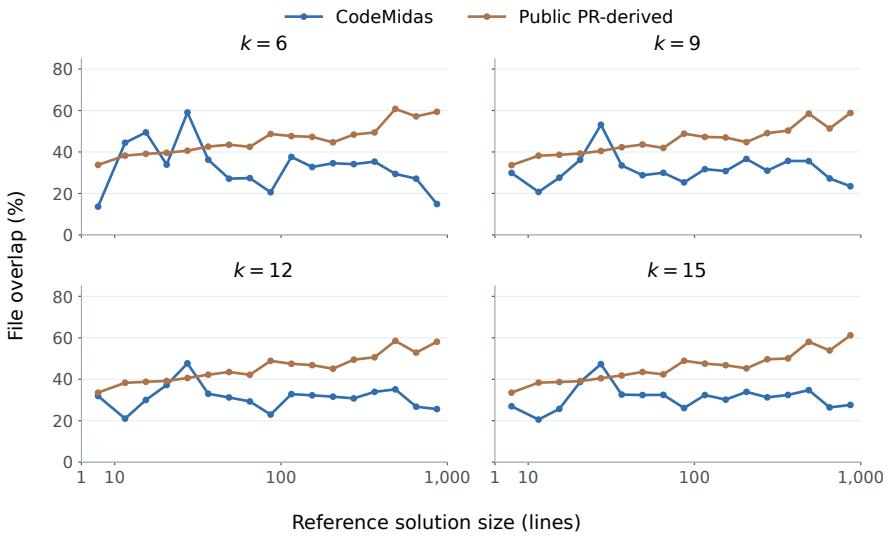

Figure 11: File overlap of CodeMidas and public PR-derived tasks at increasing k. CodeMidas uses the first k candidate positions per codebase in generation order; the public pool uses up to k available tasks in a fixed random order. Panels show weighted mean overlap by reference solution size.

B.4 Distinct Functional Requirements across Codebases

Tasks remain functionally distinct even within the same domain. Across 1,305 within-domain comparisons of 90 tasks from distinct codebases, our screening and human review of flagged candidates identify no near-duplicate pair, four partially overlapping pairs, and 1,301 distinct pairs (Table B4). This provides evidence that broader codebase coverage adds distinct functional requirements within the audited domains.

Table B4: No near-duplicate pair is found in the three audited domains. Each domain contributes one task from each of 30 sampled codebases and all 435 pairs. Counts reflect three rounds of LLM-assisted screening followed by human review of all flagged candidates.

Domain	Codebases / tasks	Pairs	Near-duplicate pairs	Partially overlapping pairs
Systems	30 / 30	435	0	2
Cloud/DevOps	30 / 30	435	0	1
Mobile	30 / 30	435	0	1

Sampling and criteria. From the final 5,545-task dataset, we select Systems, Cloud/DevOps, and Mobile: domains ranked first, eighth, and fifteenth by task count. With seed 20260925, we uniformly sample 30 codebases per domain and one task per codebase, then compare all $\binom{30}{2} = 435$ pairs per domain. Near-duplicates require essentially equivalent observable behavior after cosmetic renaming, with no substantive additional requirements; partial overlap requires shared concrete functionality plus additional behavior or constraints. Distinct pairs differ in core behavior or substantive constraints. Generic properties such as determinism alone do not establish overlap; insufficient information is a separate category.

Review procedure. We conduct three rounds of LLM-assisted screening for potential functional overlap. In each round, the model compares complete task statements and records shared requirements, differences, and supporting excerpts. We pool the pairs flagged in any round, yielding 19 unique candidates for human review. Human review identifies partial overlap in four pairs and classifies the remaining 15 as distinct. No near-duplicate pair is identified.

Even the partially overlapping pairs impose substantive differences in required behavior (Table B5). The selected domains are fixed cases, and the 1,305 pairs share 90 tasks; these counts do not estimate whole-dataset redundancy or represent independent trials.

Table B5: Shared functionality can entail different implementation requirements. Two of the four partially overlapping pairs in Table B4.

Task pair	Shared functionality	Distinct requirements
`JSTime` / `Garnet`	Stateful code loading and tracking previously loaded files	`JSTime`: ES-module linking, top-level await, JSON/WebAssembly modules. `Garnet`: Ruby `require`/`load`, load paths, and distinct repeated-execution semantics.
`OneUI` / `Graywater`	RecyclerView item counts and range notifications	`OneUI`: asynchronous whole-list differencing and suppression of stale updates. `Graywater`: multiple binders per model and flattened positions.

C Training and Evaluation Configuration

Table C6 lists the evaluation setup and shared training settings for the four task pools in Section 5.1. Groups whose 32 rollouts receive identical rewards are discarded and replaced with groups from newly sampled tasks, so each step trains on 32 groups. In the 5k run, 68.2% of sampled groups are retained.

Table C6: Training and evaluation configuration.

Setting	Value
Initial policy	MiMo-V2.5
Training task pool	5,545 CodeMidas tasks
Algorithm	GRPO
Reward	Binary verifier outcome (0 or 1)
Advantage normalization by standard deviation	Disabled (Liu et al., 2025)
Batch size	32
Rollouts per task	32
Groups with identical rewards	Discarded and replaced
Maximum prompt length (tokens)	8,192
Maximum response length (tokens)	516,096
Maximum turns per rollout	500
Maximum staleness	8
Optimizer	Adam
Learning rate	5×10^{-6}
Warmup steps	0
Adam (β_1, β_2)	(0.95, 0.95)
Adam ϵ	10^{-15}
Gradient clipping threshold	1
Weight decay	0
CodeMidas Val size	200 tasks
Evaluation attempts per task (all benchmarks)	3 (mean pass rate; avg@3)

D Additional Results on a 9B Model

We further examine whether the same CodeMidas task pool benefits a smaller initial policy. Starting from MiMo-V2.6-Distill-Qwen-9B (Xiaomi MiMo Team, 2026a), we train on the full set of 5,545 tasks using GRPO and binary execution rewards.

Training configuration. This run uses batch size 64 and 16 rollouts per task, compared with 32 and 32, respectively, for MiMo-V2.5. The remaining training hyperparameters follow Table C6.

Table D7: RL on CodeMidas with a 9B initial policy. Pass rates before and after RL for MiMo-V2.6-Distill-Qwen-9B. Gains are absolute differences in percentage points.

Evaluation	Initial policy (%)	CodeMidas RL (%)	Gain (pp)
SWE-bench Verified (OpenAI, 2024)	61.1	64.4	+3.3
SWE-bench Pro (Deng et al., 2026)	44.6	46.2	+1.6
Terminal-Bench v2.1 (Merrill et al., 2026)	37.1	42.0	+4.9
RepoZero C2Rust (Zhang et al., 2026b)	31.0	38.1	+7.1
CodeMidas Val	24.5	29.1	+4.6

Results. Table D7 reports four external benchmarks and CodeMidas Val. Terminal-Bench uses v2.1, as in the main experiment, and the additional SWE-bench evaluation uses the Verified task set (OpenAI, 2024). The 9B policy improves by 3.3 percentage points on SWE-bench Verified and 1.6 on SWE-bench Pro, alongside gains of 4.9 on Terminal-Bench v2.1 and 7.1 on RepoZero C2Rust. CodeMidas Val improves by 4.6 points. The external gains span issue repair, terminal work, and code translation, providing additional evidence that the training value of the same task pool extends across initial policies and model sizes.

E Reward Reliability During Training

We audit stratified random samples of rollout groups from training steps 1–70 to assess the reliability of the rewards used for policy updates. The audit examines whether agents obtain rewards through unintended shortcuts, whether verifier decisions agree with task requirements, and how observed reward defects differ across training windows. This complements the construction and filtering diagnostics in Appendix A by examining implementations and interactions produced during RL.

E.1 Audit Scope and Sampling

The group-level audit samples rollout groups from the 5k run (Appendix C), drawing 500 groups accepted for training and 500 groups discarded for identical rewards uniformly at random within each stratum. Discarded groups are therefore overrepresented relative to their 31.8% share of sampled groups. Table E8 reports both strata. Each group contains 32 rollouts. The audit inspects available final patches, tool-call trajectories, and verifier outcomes; reference-code comparisons use records with matched reference patches.

Table E8: Stratified group-level audit samples from training steps 1–70. Each stratum is sampled uniformly at random; each group contains 32 rollouts.

Audit sample	Sampled groups	Rollouts
Accepted for training	500	16,000
Discarded (identical rewards)	500	16,000
Total	1,000	32,000

A separate human review uses stratified random samples from groups accepted for training (Section E.3). It assesses agreement with task requirements. Group-level scans and submission-level reviews therefore have distinct coverage and denominators.

E.2 Reward Bypasses in Training Rollouts

Among verifier-accepted submissions in the sampled training groups, the proportion involving confirmed residual or undeleted reference code is **0.1%**. We find no confirmed instance of the targeted submission-level exploit patterns among the reviewed matches, and no confirmed access to hidden test contents in the examined trajectories.

Submission-level and environment checks. We scan added patch lines for known exploit patterns (Baker et al., 2025), including always-true equality overrides, exits before assertions, test-report hooks, test skipping, and changes to test configuration. Generated and third-party directories are excluded. Matched cases are reviewed by task, prioritizing tasks with the most matches in each category; reviewed matches correspond to legitimate code, including help-command exits and checks written by the agent. We also inspect access to codebase history and verifier-related artifacts. Network access is disabled, and reviewed history accesses expose the starting commit or commits created by the solver.

Reference-code checks. We flag reads outside the source tree whose outputs contain at least half of the nontrivial added lines in the matched reference patch. We also examine reads before the first edit to check whether symbols added by the reference patch are already defined, identifying undeleted implementations. Suspected cases are checked for residual or undeleted reference code.

We distinguish environment exposure from rewards used for optimization. All-pass groups are discarded by GRPO and contribute no policy update. The reported 0.1% incidence concerns passing submissions within the sampled groups accepted for training. Findings about the targeted exploit patterns and hidden-test access are restricted to the evidence examined by these checks.

E.3 Human Review of Verifier Decisions

Human-reviewed samples. We manually review stratified random samples of verifier-accepted and verifier-rejected submissions from groups accepted for training, using early (steps 1–10) and later (steps 51–61) windows. Sampling uses a fixed random seed and at most two submissions per task in each stratum. Tasks flagged in the environment audit are handled separately and excluded from this sampling pool. The human-reviewed sample contains 130 early and 135 later accepted submissions, and 146 early and 149 later rejected submissions.

Review criteria. For accepted submissions, a main-usage defect requires an incorrect observable result in a conventional use of a central operation described or illustrated by the task. Secondary behavior, unusual inputs, and platform or build issues are outside this criterion. For rejected submissions, an unsupported rejection requires no established main-usage defect and a failing check that demands behavior absent from the statement. Violations of any explicit requirement remain valid rejections. Environment failures and uncertain cases are not counted as unsupported rejections. Uncertain cases remain in their stratum denominators but do not enter the confirmed-error counts.

Table E9: Human review across training windows. Accepted and rejected submissions have separate denominators; all reviewed submissions in each stratum are included. Confidence intervals resample whole tasks.

Measure	Early (steps 1–10)	Later (steps 51–61)
Main-usage defects among accepted submissions	8/130 (6.2%)	6/135 (4.4%)
95% confidence interval	2.3–10.7%	1.4–8.3%
Unsupported rejections among rejected submissions	17/295 (5.8%, pooled)	

Table E9 reports main-usage defects in 6.2% of early and 4.4% of later accepted submissions. The later-minus-early difference is −1.7 percentage points, calculated from unrounded proportions using 2,000 bootstrap resamples of whole tasks. The interval includes zero, leaving the direction of change uncertain. The windows contain different task samples. Across both windows, 17 of 295 rejected submissions (5.8%) are judged to fail checks unsupported by the statement, including checks on unspecified formatting or code style.

F Behavioral Measures and Aggregation

Codebase exploration. We count distinct read/search requests before the first codebase edit. Reads are deduplicated by file and line range, and searches by query, scope, and options. Equivalent requests through dedicated tools or shell commands count once.

Code drafting. For each `Write`/`Edit` payload, we sample distinct 16-character fragments at a stride of four characters. The drafting ratio is the percentage found anywhere in reasoning before the corresponding write, computed from summed matched and total fragment counts across write actions.

Self-verification. We count distinct verification commands after the final codebase edit, including project tests, inline checks, and local program runs; repeated commands count once. The success association uses a separate indicator of whether the agent wrote and executed a check.

Aggregation. Training means use the Early/Late stages in Table 2, computed over all sampled rollouts, including discarded groups. External summaries average the first and last three observed checkpoint means (Table 3). Each behavioral measure excludes undefined values. Interaction length counts assistant turns over all scored rollouts.

Success associations. On CodeMidas Val, we compute pass-rate differences between behavioral groups within each task and checkpoint where both groups are present. We average checkpoint differences within each task, then average equally across tasks. Exploration and drafting use checkpoint-median splits.