WHEN REQUIREMENTS INTERACT: SCALING LONG-HORIZON CODING VIA INTERDEPENDENT SPECIFICATIONS

Anonymous authors
Paper under double-blind review

ABSTRACT

To succeed in real-world long-horizon software development, coding agents must satisfy multi-dimensional task specifications, such as interface compatibility and error handling, and do so without breaking existing dependencies. However, current works lack specification in both quantity and diversity, often emphasizing primary goals while ignoring complex constraints. Synthesizing tasks to bridge this gap presents two challenges: naively adding specifications risks reducing tasks to trivial checklists rather than compelling joint reasoning, and verifying these subtle interactions risks overlooking hidden flaws or rejecting valid alternatives. To address these challenges, we introduce CODESPECWEAVE, a framework for synthesizing tasks with rich, interdependent specifications. Grounded in real developer seeds, a synthesis agent iteratively develops both coupled task requirements and executable tests, while a review agent audits solution trajectories from multiple models to guarantee test reliability. Using this pipeline, we construct 4,670 robust training tasks across 2,865 repositories. Applying reinforcement learning on a subset of these tasks enables MiMo-V2.6-Flash-SFT to achieve 63.36% on DeepSWE-v1.1, substantially outperforming existing baselines. Furthermore, we curate CODESPECWEAVE-BENCH, a rigorous benchmark comprising 100 tasks and 1,044 specifications, which clearly separates frontier models and reveals significant headroom in handling interdependent engineering constraints.

1 INTRODUCTION

Recent advances in large language models (LLMs) have enabled coding agents to explore repositories, modify code, execute tests, and iteratively refine their solutions (Yang et al., 2024). As these agents navigate the increasing complexity of real-world software architectures, they must implement changes without breaking behavior that other components already depend on (Jimenez et al., 2024; Deng et al., 2026). In practice, these expectations span multiple requirements, such as interface compatibility, state consistency, and error handling (Han et al., 2024; Ruan et al., 2025). Together, they form concrete task specifications that an agent must satisfy throughout exploration, coding, and testing.

Developing this capability requires a large number of executable tasks that encompass these diverse engineering requirements. Existing works typically construct executable tasks by mining issues and pull requests (Pan et al., 2025), with limited attention to the role and properties of task specifications. Our empirical analysis reveals that existing data fall significantly short in both specification density and structural diversity (Figure 1, top), often emphasizing the primary development goal while leaving compatibility, state consistency, and error handling unaddressed. Consequently, agents trained on such partial specifications often fail in practice, as they struggle to satisfy the intricate constraints inherent in real-world repositories. This deficiency underscores the need for a scalable pipeline to synthesize tasks grounded in realistic and diverse specifications.

However, synthesizing such tasks presents two connected challenges. First, purely adding specifications may inadvertently reveal solution paths (Kim et al., 2026) or reduce the task to a checklist of independent edits. Specification design should therefore organize requirements into interdependent specifications where individual implementation choices constrain one another, compelling agents

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.

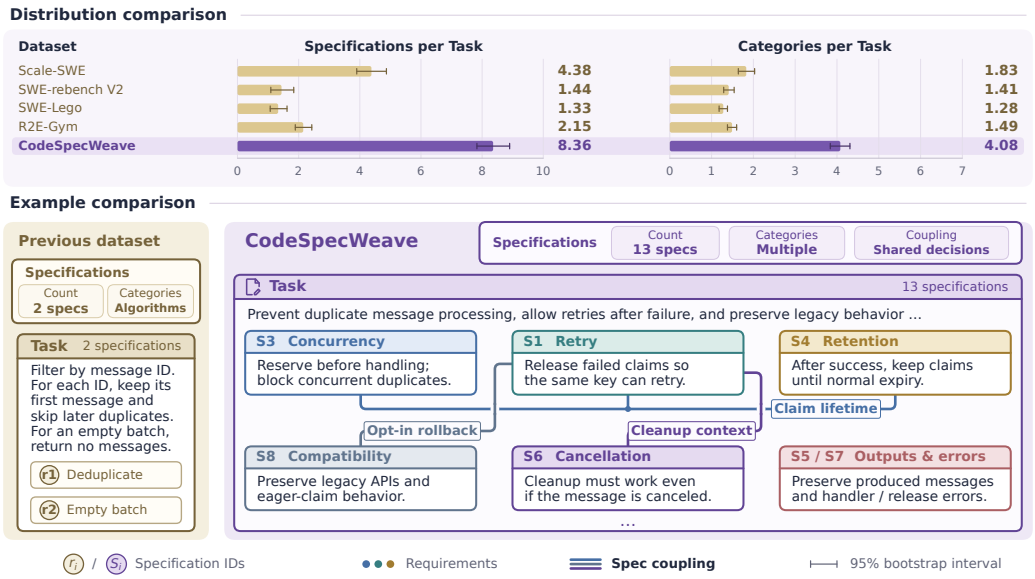


Figure 1: **Comparison of task specifications across datasets. Top:** Mean extracted specifications and distinct behavioral categories per task across existing SWE training datasets and CODESPECWEAVE, with 95% bootstrap confidence intervals. Category counts use the seven reporting categories in Table 5. **Bottom:** Qualitative comparison of task structures. The synthesized example combines control flow and ordering, state and side effects, and errors and recovery, with shared decisions on claim lifetime and cleanup context. Labels such as concurrency, retry, and compatibility identify concrete requirements or auxiliary concerns.

to coordinate across requirements through joint reasoning to achieve globally coherent implementations (Figure 1, bottom). Second, verifying such interdependent specifications is uniquely challenging: the subtle interactions among requirements make it easy for tests either to miss hidden flaws (Sun et al., 2026; Tan et al., 2026) or to reject valid solutions (Chowdhury et al., 2024). Reliable verification requires strict alignment with both individual requirements and their interactions, while accommodating different implementations that satisfy them.

To this end, we introduce CODESPECWEAVE, a framework for synthesizing coding tasks with rich, interdependent specifications. Grounded in specification seeds derived from real developers, a synthesis agent explores the repository context and develops the task requirements and executable test suites together in an iterative loop, intentionally designing coupled constraints that require simultaneous satisfaction while refining the tests to strictly align with them. To ensure test reliability, a review agent further examines the solution trajectories and patches from multiple models, filtering out tests that accept incomplete implementations or reject valid alternatives. Using this pipeline, we construct 4,670 training tasks across 2,865 repositories. Applying reinforcement learning on a subset of these tasks achieves 63.36% on DeepSWE-v1.1 for MiMo-V2.6-Flash-SFT, substantially outperforming other training baselines. In addition, we curate CODESPECWEAVE-BENCH through careful manual review, compiling 100 tasks with 1,044 specifications to evaluate coding agents under interdependent specifications. Benchmarking leading models clearly separates their capabilities, revealing shared failure modes and significant room for improvement.

Our contributions are as follows:

- We introduce CODESPECWEAVE, a synthesis framework that builds on real developer specification seeds to generate coding tasks with interdependent specifications and reliable verifiers.
- We construct a scalable training set of 4,670 tasks across 2,865 repositories and demonstrate its utility by training MiMo-V2.6-Flash-SFT with reinforcement learning on a subset of these tasks, achieving 63.36% on DeepSWE-v1.1, outperforming other training baselines.

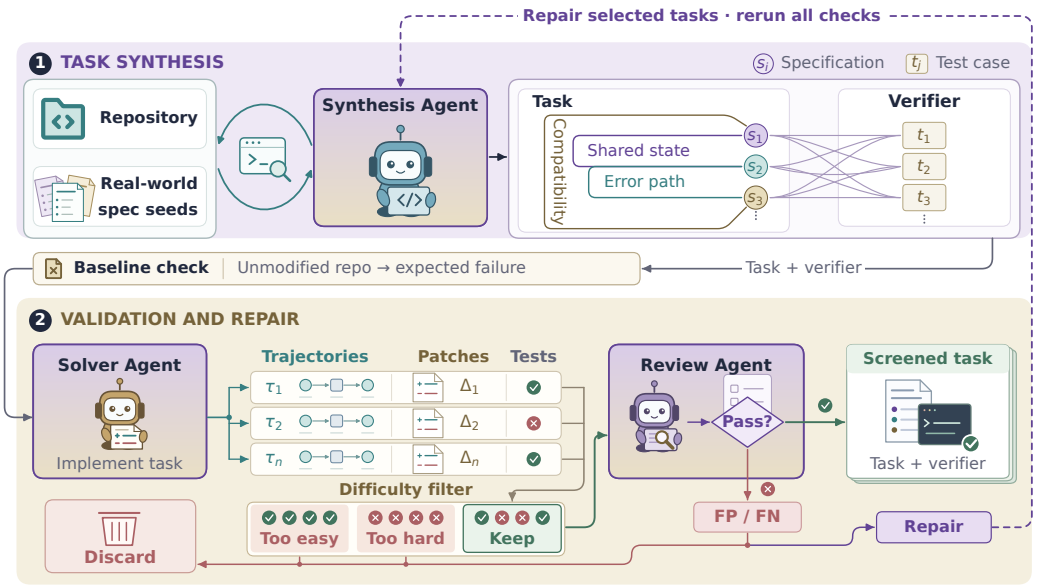


Figure 2: **Overview of CodeSpecWeave.** The synthesis agent repeatedly consults specification seeds, explores the repository, and runs generated tests to refine a task and its verifier. Schematic links between specifications and test cases illustrate many-to-many coverage. Solver trials and trajectory review then assess difficulty and query–test alignment. Accepted task–verifier pairs are retained; selected candidates with false positives or false negatives are repaired and revalidated.

- We present CODESPECWEAVE-BENCH, a benchmark comprising 100 tasks with 1,044 specifications. Evaluations across leading coding agents reveal notable performance gaps and substantial headroom under interdependent requirements.

2 CODESPECWEAVE

CodeSpecWeave transforms requirements observed in real software development into coding tasks, addressing two connected challenges: compelling agents toward holistic exploration and joint reasoning under rich specifications, and scaling reliable test verification aligned with stated requirements. We first introduce the task formulation (Section 2.1) and design principle (Section 2.2). The framework consists of two core stages (Figure 2): *task synthesis* constructs interdependent requirements and linked tests guided by specification seeds (Section 2.3); *validation and repair* reviews trajectories generated by multiple model trials to identify and repair misalignments between queries and tests (Section 2.4).

2.1 PRELIMINARIES AND PROBLEM FORMULATION

We represent a task as a tuple $\mathcal{T} = (\mathcal{R}, \mathcal{Q}, \mathcal{S}, \mathcal{E})$. Here, $\mathcal{R}$ is the code repository, $\mathcal{Q}$ is the natural-language request provided to the solver, $\mathcal{S} = \{s_1, \ldots, s_m\}$ is the set of specifications embedded within $\mathcal{Q}$ that a valid patch must satisfy, and $\mathcal{E}$ is an execution environment equipped with a verifier V. Given $\mathcal{Q}$ and $\mathcal{R}$, a solver produces a patch Δ through repository interactions. We write $s_i(\Delta) = 1$ when the patched repository satisfies specification s_i, and 0 otherwise. The set of valid patches is defined as:

$$\mathcal{F}(\mathcal{T}) = \{\Delta : \ s_i(\Delta) = 1 \text{ for all } i\}. \tag{1}$$

In practice, the verifier V checks whether $\Delta \in \mathcal{F}(\mathcal{T})$ by executing test suites.

2.2 Design Principle: Interdependent Specifications

Integrating rich specifications into coding tasks requires organizing how requirements relate to one another, rather than merely accumulating them. In our initial synthesis attempts, we observed that when relying solely on specification count, the synthesized tasks remained trivial: language models achieved high success simply by satisfying each requirement in isolation. This observation motivates distinguishing *additive specifications*, whose requirements admit largely separate decisions, from *interdependent specifications*, where a choice made to satisfy one requirement constrains the feasible options for another.

Formally, let design variables $d = (d_1, \ldots, d_k)$ denote the engineering choices permitted by specification set $\mathcal{S}$ within feasible space $\mathcal{F}_d$. An idealized additive task factorizes independently as $\mathcal{F}_d = \prod_j \mathcal{F}_j$, reducing to a step-by-step checklist where each choice can be resolved greedily in isolation. In contrast, interdependent specifications break this factorization ($\mathcal{F}_d \subsetneq \prod_j \mathcal{F}_j$), such that selecting one choice strictly restricts the feasible alternatives for others. Because local edits cannot be decoupled, the solver must reason across all requirements simultaneously, presenting much more challenging tasks for existing agents.

2.3 Task Synthesis

To reflect real-world developer requirements, we extract specification seeds from open-source repositories by mining commit histories, issue discussions, and pull requests. After filtering out conversational chatter and raw code fragments, normalized deduplication yields 104,182 candidate specification units across six natural languages. These seeds inform a compact specification reference containing examples, definitions, and boundary guidance for task synthesis.

Given a repository $\mathcal{R}$, specification seeds, and prompt instructions to construct interdependent specifications (§2.2), the synthesis agent explores the codebase to formulate development tasks. It combines complementary requirements, anchoring them to the same state transitions or core components so that satisfying one requirement constrains the implementation of others. For instance, a task may couple input validation, transactional rollback, and backward compatibility for existing readers, ensuring that satisfying any single requirement in isolation still violates the broader contract. The agent iteratively drafts the task request $\mathcal{Q}$, fine-grained specifications $\mathcal{S}$, and corresponding test cases $\mathcal{A}$. These tests map flexibly to specifications, allowing multiple checks to validate a single requirement or an integration test to cover several obligations simultaneously. Throughout this process, the agent executes generated tests directly in the repository and refines them based on runtime feedback.

2.4 Validation and Repair

Simultaneously generating interdependent specifications and tests does not guarantee capturing the intended requirement boundary. Because multiple requirements closely interact, tests often fail to check flaws that arise only when the requirements interact at runtime, allowing incorrect solutions to pass (Sun et al., 2026; Tan et al., 2026). At the same time, verifying coupled requirements easily causes tests to assume a specific implementation, rejecting valid solutions that satisfy the full specification (Chowdhury et al., 2024). Our pilot generation reflects this difficulty: automated reviews identified alignment flaws in 2,267 of 3,067 candidate tasks (73.9%; Table 1), confirming that complex specifications make reliable test generation exceedingly hard.

Table 1: Test alignment breakdown on 3,067 candidates.

Issue	Tasks	%
False positives only	1,548	50.5
False negatives only	426	13.9
Both	293	9.6
None observed	800	26.1
Total with issues	2,267	73.9

Because these flaws only appear when code actually runs against the requirements, inspecting static code or tests alone is insufficient. Instead, we collect candidate patches Δ and their execution trajectories τ from independent coding models across multiple attempts, filtering for tasks with mixed solver success to keep challenging yet solvable tasks. Comparing these execution trajectories with test results reveals errors that are invisible during generation: runs that violate multiple requirements expose missing checks, while valid runs that fail indicate tests that enforce unintended restrictions. A review agent inspects the full tuple $(\mathcal{Q}, \mathcal{A}, \Delta, \tau)$ to iteratively reconcile the tests with the inter-

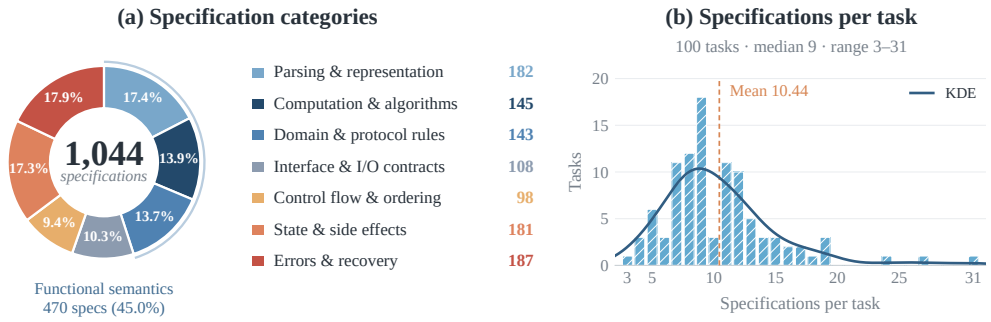


Figure 3: **Specification distribution in CodeSpecWeave-Bench.** (a) The five behavioral categories in Table 5, with functional semantics expanded into three blue subcategories. Each of the 1,044 specifications contributes once to one of the seven segments. (b) Specifications per task across 100 tasks (mean: 10.44).

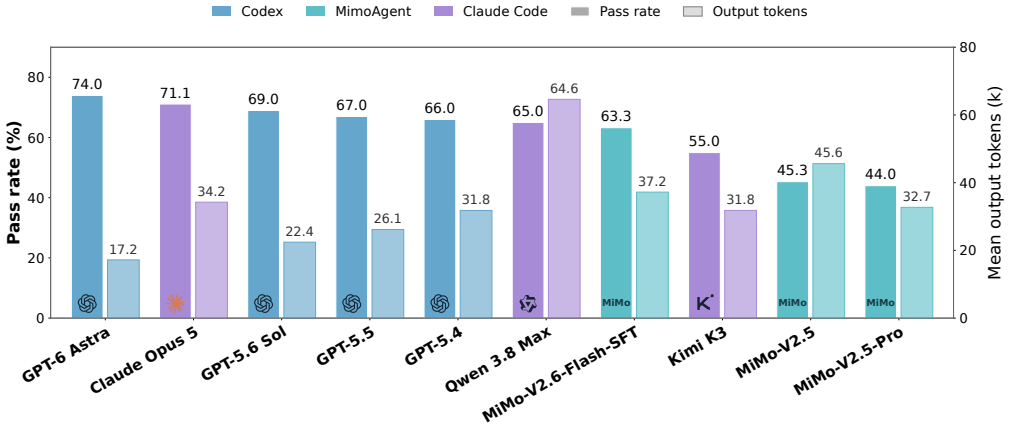


Figure 4: **Pass rates and mean reported output tokens on CodeSpecWeave-Bench.** Darker bars show pass rates (left axis); lighter bars show output-token means. Colors identify agent frameworks.

dependent specifications. In one generation batch of 14,905 initial tasks, difficulty filtering retained 3,275 candidates, from which this review approved 766 verified tasks.

For selected candidates with identified false positives or false negatives, a repair agent uses the review findings and execution trajectories to revise the task request or verifier. Each repaired task–verifier pair then undergoes execution checks, fresh solver trials, and alignment review before acceptance.

Using the full CodeSpecWeave pipeline, we synthesize 4,670 training tasks across 2,865 repositories for reinforcement learning. For evaluation, we curate CodeSpecWeave-Bench, a benchmark of 100 manually reviewed tasks. Figure 3 summarizes its 1,044 recorded specifications by behavioral category and task, with a mean of 10.44 specifications per task.

3 Experiments

3.1 Evaluation on CodeSpecWeave-Bench

Evaluation setup. We evaluate GPT-5.4, GPT-5.5, GPT-5.6 Sol, and GPT-6 Astra (OpenAI, 2026a;b;c;d) with Codex (OpenAI); Claude Opus 5, Kimi K3, and Qwen 3.8 Max (Anthropic, 2026; Moonshot AI, 2026; Qwen Team, 2026) with Claude Code (Anthropic); and MiMo-V2.5, MiMo-

V2.5-Pro, and MiMo-V2.6-Flash-SFT (Xiaomi MiMo Team, 2026a;b; LLM-Core Xiaomi, 2026) with MimoAgent.

Results. Figure 4 shows performance across representative agents on CodeSpecWeave-Bench. Scores range widely from 44.0% to 74.0%, demonstrating that the benchmark clearly distinguishes model capabilities. GPT-6 Astra (74.0%) and Claude Opus 5 (71.1%) achieve the highest success, followed by Qwen 3.8 Max (65.0%) and MiMo-V2.6-Flash-SFT (63.3%). The latter substantially narrows the gap to frontier proprietary models. However, even the strongest configuration fails on 26 of the 100 tasks, showing that satisfying interdependent specifications remains challenging. Mean output tokens range from 17.2k to 64.6k under the respective evaluation settings. GPT-6 Astra achieves the top score with the shortest output length (17.2k tokens), whereas Qwen 3.8 Max requires substantially longer rollouts (64.6k tokens) despite achieving a lower pass rate.

3.2 Reinforcement learning on CodeSpecWeave

3.2.1 Training and evaluation setup

Models. We apply reinforcement learning to two Mixture-of-Experts (MoE) models, MiMo-V2.5 (Xiaomi MiMo Team, 2026a) and MiMo-V2.6-Flash-SFT (LLM-Core Xiaomi, 2026), each with 310B total and 15B active parameters.

Training datasets. We compare three RL datasets: CodeSpecWeave, SWE-PI, and MiMo Coding. CodeSpecWeave contains 4,670 tasks across 2,865 repositories, pairing interdependent development requirements with executable verifiers. MiMo-V2.5 uses the full set, while MiMo-V2.6-Flash-SFT uses a 2,000-task subset from 1,504 repositories. SWE-PI contains 30,128 repository-level tasks curated from diverse SWE environments (Auras & Prime Intellect Team, 2026), combining Scale-SWE (Zhao et al., 2026), SWE-Lego (Tao et al., 2026), SWE-rebench V2 (Badertdinov et al., 2026), and Multi-SWE-RL (Zan et al., 2025). MiMo Coding (LLM-Core Xiaomi, 2026) is curated from diverse collection pipelines, including issue-driven development, everyday development workflows, functionality implementation, and long-horizon engineering.

Evaluation benchmarks. We evaluate on four benchmarks that cover specification-rich development and repository-level issue resolution. CodeSpecWeave-Bench contains 100 tasks from 100 repositories with interdependent requirements and executable tests. DeepSWE-v1.1 contains 113 original, long-horizon engineering tasks and grades submitted code in isolated environments (Huang et al., 2026). SWE-bench Verified comprises 500 human-validated GitHub issues from Python repositories (Jimenez et al., 2024; Chowdhury et al., 2024). SWE-bench Pro extends issue resolution to challenging, multilingual development tasks, and we use its 731-task public set (Deng et al., 2026).

3.2.2 Evaluation results

Table 2 compares performance across four benchmarks. Across both MiMo-V2.5 and MiMo-V2.6-Flash-SFT backbones, CodeSpecWeave yields the highest average scores (50.23% and 67.67%), outperforming other training datasets. The improvements are especially clear on CodeSpecWeave-Bench and DeepSWE. On CodeSpecWeave-Bench, CodeSpecWeave consistently achieves the top scores across both base models (54.67% and 68.67%). The advantage is even more pronounced on DeepSWE, which presents challenging long-horizon software engineering tasks. While other datasets show limited gains or even negative transfer (e.g., SWE-PI on MiMo-V2.5), CodeSpecWeave substantially boosts scores to 24.34% and 63.36%. Meanwhile, CodeSpecWeave maintains competitive performance on SWE-bench Verified and SWE-bench Pro.

Table 3: Mean thinking tokens and code changes per trajectory.

Measure	Vanilla	CodeSpecWeave
MiMo-V2.5		
Thinking tokens (k)	29.6	115.4
Modified files	4.03	4.44
Changed lines	435	693
MiMo-V2.6-Flash-SFT		
Thinking tokens (k)	23.6	101.5
Modified files	3.84	8.51
Changed lines	333	808

Table 2: Task success rates at the selected checkpoints (%, higher is better). Avg is the unweighted mean across the four benchmarks.

Training Settings	CodeSpecWeave-Bench	DeepSWE	SWE-bench Verified	SWE-bench Pro	Avg
MiMo-V2.5					
w/ Vanilla	45.33	7.74	68.73	49.64	42.86
w/ SWE-PI	43.33	6.49	68.67	49.29	41.95
w/ MiMo Coding	48.00	17.11	74.47	54.17	48.44
w/ CodeSpecWeave	54.67	24.34	70.93	50.98	50.23
MiMo-V2.6-Flash-SFT					
w/ Vanilla	63.27	51.93	72.60	55.69	60.87
w/ SWE-PI	67.00	50.29	77.20	59.51	63.50
w/ MiMo Coding	68.35	53.10	77.80	59.60	64.71
w/ CodeSpecWeave	68.67	63.36	77.47	61.19	67.67

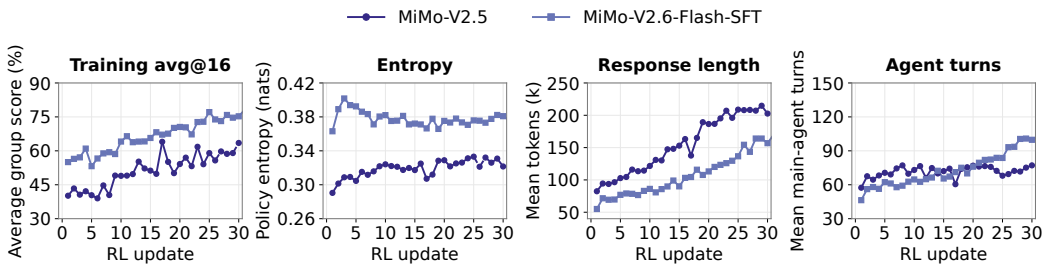


Figure 5: Training dynamics through RL update for the two CodeSpecWeave runs. From left to right: mean task-group score, policy entropy, response length, and main-agent turns.

Model behavior. After CodeSpecWeave training, MiMo-V2.5 generates 3.9 times as many thinking tokens and MiMo-V2.6-Flash-SFT 4.3 times as many (Table 3). Their editing patterns differ: MiMo-V2.5 makes more extensive revisions within a small set of files, while Flash-SFT also expands the number of files modified. For example, in mruby, base MiMo-V2.5 solutions implement exception injection but leave the target fiber alive after an unhandled exception. After training, the solutions correctly handle cleanup and termination and pass the tests (Appendix D.1).

3.2.3 Training Dynamics

Figure 5 shows the RL updates of CodeSpecWeave across both backbones. Within this window, average task scores increase, with MiMo-V2.6-Flash-SFT rising from 55.02% to 75.26%. These gains accompany longer responses and more agent turns; mean turns for MiMo-V2.6-Flash-SFT rise from 46.37 to 99.77. Policy entropy remains between 0.29 and 0.40 across both runs.

4 Related Work

Coding Data Synthesis. Magicoder (Wei et al., 2024) generates coding problems and solutions from open-source code snippets, while KodCode (Xu et al., 2025) draws on programming exercises, algorithm implementations, and library documentation. At the repository level, R2E-Gym (Jain et al., 2025) derives tasks from commits through test generation and back-translation, SWE-smith (Yang et al., 2026) modifies working code to create repair tasks, and SWE-Flow (Zhang et al., 2025) uses unit tests and runtime dependencies to construct incremental development tasks. Other methods expand how task content is generated: SWE-Mirror (Wang et al., 2025) transfers issue semantics across repositories, ADR (Zheng et al., 2026) recombines semantic elements extracted from seed problems, and TaskPilot (Kim et al., 2026) uses solver feedback to refine jointly generated tasks, reference patches, and tests. CodeSpecWeave makes the coverage and interaction

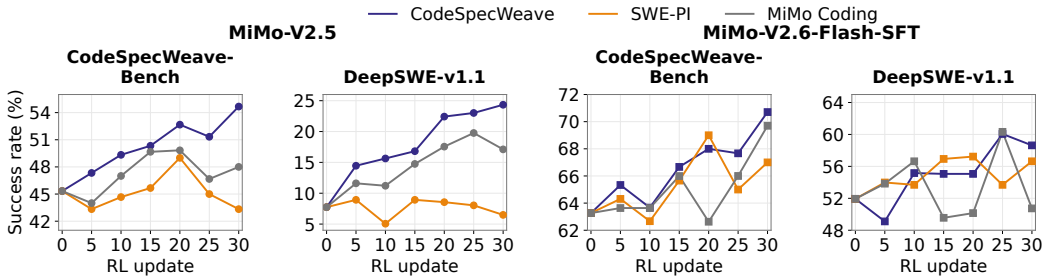


Figure 6: Learning curves for CodeSpecWeave, SWE-PI, and MiMo Coding, trained from MiMo-V2.5 (left) and MiMo-V2.6-Flash-SFT (right).

of engineering requirements explicit objectives of task synthesis. Specification seeds guide the selection of complementary requirements, while repository exploration connects them through shared state transitions and interfaces. This allows us to construct training tasks that exercise joint reasoning about functionality, compatibility, and failure recovery within the same code change.

Software Specifications. Software specifications make the behaviors and constraints of a development task explicit. ArchCode (Han et al., 2024) organizes functional and non-functional requirements from problem descriptions, while SpecRover (Ruan et al., 2025) infers intended behavior from issue descriptions and repository context. REAgent (Kuang et al., 2026) produces structured requirements that include root-cause analysis and proposed edits, while CodeSpec (Wang et al., 2026) constructs cross-component functional chains to guide feature development. Specifications also structure evaluation and learning: DevAI (Zhuge et al., 2025) organizes development requirements as dependency graphs, SWE-AGI (Zhang et al., 2026) evaluates system construction from standards and RFCs, and SpecCoder (Li et al., 2026) trains models to produce specification analyses and code through supervised fine-tuning and reinforcement learning. CodeSpecWeave specifies interacting behavioral obligations without supplying a repair or architectural plan. For example, validation, rollback, and compatibility requirements constrain the behavior of the same update operation. The solver must explore the repository and choose a design that satisfies these obligations together, making both exploration and requirement coordination part of the training task.

5 Conclusion and Limitations

We introduced CodeSpecWeave, a framework for synthesizing executable coding tasks with interdependent specifications in real repositories. Starting from specification seeds drawn from developer interactions, a synthesis agent jointly refines task requirements and tests, while a review agent examines solutions from multiple models to identify verification errors. The resulting training set contains 4,670 tasks across 2,865 repositories. Reinforcement learning on a subset of these tasks raises MiMo-V2.6-Flash-SFT's DeepSWE-v1.1 success rate from 51.93% to 63.36%, with gains over the base model on all four evaluated benchmarks. We also introduced CodeSpecWeave-Bench, comprising 100 manually reviewed tasks and 1,044 specifications. Evaluations reveal substantial performance differences across agent configurations and remaining challenges in satisfying interacting requirements. Together, these results support the use of tasks with interdependent specifications for coding-agent training and evaluation.

Our approach has three practical limitations. First, using the synthesized tasks to improve model performance through RL requires substantial computational resources for training experiments. Second, each task has an associated container image and runs in a Kubernetes pod. Training and evaluation therefore require infrastructure to manage many containers concurrently. Third, producing high-quality tasks relies on strong models and multiple rounds of filtering, which discard a large fraction of generated candidates.

AI USE STATEMENT

We used generative AI to assist with task synthesis and validation, data analysis, and manuscript polishing.

REFERENCES

Anthropic. Claude Code Overview. URL https://code.claude.com/docs/en/overview.

Anthropic. System Card: Claude Opus 5. System card, July 2026. URL https://www.anthropic.com/claude-opus-5-system-card.

Daniel Auras and Prime Intellect Team. Scaling Agentic RL: 365,000+ Environments for SWE, Terminal, and Search. Prime Intellect blog, July 2026. URL https://www.primeintellect.ai/blog/scaling-agentic-rl.

Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. SWE-rebench v2: Language-agnostic SWE task collection at scale. In *Proceedings of ICML*, 2026. URL https://openreview.net/forum?id=UCAda9kS57.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench Verified. OpenAI blog, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/. Published August 13, 2024.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa R Kundurthy, Sean M. 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 ICML*, 2026. URL https://openreview.net/forum?id=uEVTdoAbnK.

Hojae Han, Jaejin Kim, Jaeseok Yoo, Youngwon Lee, and Seung-won Hwang. ArchCode: Incorporating software requirements in code generation with large language models. In *Proceedings of ACL*, pp. 13520–13552, 2024. URL https://aclanthology.org/2024.acl-long.730/.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *arXiv preprint arXiv:2607.07946*, 2026. URL https://arxiv.org/abs/2607.07946.

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

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 *Proceedings of ICLR*, volume 2024, pp. 54107–54157, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/file/edac78c3e300629acfe6cbe9ca88fb84-Paper-Conference.pdf.

Minseon Kim, Zhengyan Shi, Emiliano Penaloza, Christopher Cui, Roger Creus Castanyer, Maryam Hashemzadeh, Isadora White, Jonathan Light, Jeonghye Kim, Matheus Pereira, et al. Frognano: Training a 4b coding agent via online task synthesis. *arXiv preprint arXiv:2609.07925*, 2026. URL https://arxiv.org/abs/2609.07925.

Shiqi Kuang, Zhao Tian, Kaiwei Lin, Chaofan Tao, Shaowei Wang, Haoli Bai, Lifeng Shang, and Junjie Chen. Reagent: Requirement-driven llm agents for software issue resolution. *arXiv preprint arXiv:2604.06861*, 2026. URL https://arxiv.org/abs/2604.06861.

Yixuan Li, Mingxuan Huang, Jiajing Wang, Weidong Yang, Xinyi Liu, Ben Fei, and Lipeng Ma. Speccoder: Specification-aware code generation with curriculum dual-task reinforcement learning. *arXiv preprint arXiv:2609.06041*, 2026. URL https://arxiv.org/abs/2609.06041.

LLM-Core Xiaomi. MiMo-V2.6: Scaling Reinforcement Learning Towards Self-Improvement. Technical report, 2026. URL https://huggingface.co/XiaomiMiMo/MiMo-V2.6-Flash-RL/resolve/main/MiMo_V2_6_technical_report.pdf.

Moonshot AI. Kimi K3. Hugging Face model card, 2026. URL https://huggingface.co/moonshotai/Kimi-K3.

OpenAI. Codex CLI. URL https://learn.chatgpt.com/docs/codex/cli.

OpenAI. GPT-5.4 Thinking System Card. System card, March 2026a. URL https://deploymentsafety.openai.com/gpt-5-4-thinking.

OpenAI. GPT-5.5 System Card. System card, April 2026b. URL https://deploymentsafety.openai.com/gpt-5-5.

OpenAI. GPT-5.6 System Card. System card, July 2026c. URL https://deploymentsafety.openai.com/gpt-5-6.

OpenAI. GPT-6 Astra System Card. System card, September 2026d. URL https://deploymentsafety.openai.com/gpt-6-astra.

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 ICML*, volume 267, pp. 47717–47737, 2025. URL https://proceedings.mlr.press/v267/pan25g.html.

Prime Intellect. R2E-Gym-Subset-Verified. Hugging Face dataset card, 2026a. URL https://huggingface.co/datasets/PrimeIntellect/R2E-Gym-Subset-Verified.

Prime Intellect. Scale-SWE-Verified. Hugging Face dataset card, 2026b. URL https://huggingface.co/datasets/PrimeIntellect/Scale-SWE-Verified.

Prime Intellect. SWE-Lego-Real-Data-Verified. Hugging Face dataset card, 2026c. URL https://huggingface.co/datasets/PrimeIntellect/SWE-Lego-Real-Data-Verified.

Prime Intellect. SWE-rebench-V2-Filtered-Verified. Hugging Face dataset card, 2026d. URL https://huggingface.co/datasets/PrimeIntellect/SWE-rebench-V2-Filtered-Verified.

Qwen Team. Qwen3.8-Max: A New Bar for Coding and Cowork. Qwen blog, August 2026. URL https://qwen.ai/blog?id=qwen3.8.

Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. Specrover: Code intent extraction via llms. In *Proceedings of ICSE*, pp. 963–974, 2025.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024. URL https://arxiv.org/abs/2402.03300.

Yuxuan Sun, Yuze Zhao, Yufeng Wang, Yao Du, Zhiyuan Ma, Jinbo Wang, Mengdi Zhang, Kai Zhang, and Zhenya Huang. SWE-mutation: Can LLMs generate reliable test suites in software engineering? In *Findings of ACL*, pp. 39651–39674, 2026. URL https://aclanthology.org/2026.findings-acl.1976/.

Yuhao Tan, Zhibang Yang, Fangkai Yang, Yuan Yao, Yu Kang, Lu Wang, Pu Zhao, Xin Zhang, Xiaoxing Ma, Qingwei Lin, et al. Beyond fail-to-pass: Iterative hardening of co-generated bug reproduction tests and fixes. *arXiv preprint arXiv:2607.19843*, 2026. URL https://arxiv.org/abs/2607.19843.

Chaofan Tao, Jierun Chen, Yuxin Jiang, Kaiqi Kou, Shaowei Wang, Ruoyu Wang, Xiaohui Li, Sidi Yang, Yiming Du, Jianbo Dai, et al. 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.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. Swe-mirror: Scaling issue-resolving datasets by mirroring issues across repositories. *arXiv preprint arXiv:2509.08724*, 2025. URL https://arxiv.org/abs/2509.08724.

Peiding Wang, Li Zhang, Fang Liu, Taichuan Li, and Yinghao Zhu. Codespec: Dual executable specifications for agentic long-horizon feature development. *arXiv preprint arXiv:2607.26777*, 2026. URL https://arxiv.org/abs/2607.26777.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-instruct. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of ICML*, volume 235, pp. 52632–52657, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/wei24h.html.

Xiaomi MiMo Team. MiMo-V2.5. Hugging Face model card, 2026a. URL https://huggingface.co/XiaomiMiMo/MiMo-V2.5.

Xiaomi MiMo Team. MiMo-V2.5-Pro. Hugging Face model card, 2026b. URL https://huggingface.co/XiaomiMiMo/MiMo-V2.5-Pro.

Zhangchen Xu, Yang Liu, Yueqin Yin, Mingyuan Zhou, and Radha Poovendran. KodCode: A diverse, challenging, and verifiable synthetic dataset for coding. In *Findings of ACL*, pp. 6980–7008, 2025. URL https://aclanthology.org/2025.findings-acl.365/.

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 *Proceedings of NeurIPS*, volume 37, pp. 50528–50652, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

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 *Proceedings of NeurIPS Datasets and Benchmarks Track*, 2026. URL https://openreview.net/forum?id=63iVrXc8cC.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving, 2025. URL https://arxiv.org/abs/2504.02605.

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 ICML*, volume 267, pp. 76518–76540. PMLR, 2025. URL https://proceedings.mlr.press/v267/zhang25cn.html.

Zhirui Zhang, Hongbo Zhang, Haoxiang Fei, Zhiyuan Bao, Yubin Chen, Zhengyu Lei, Ziyue Liu, Yixuan Sun, Mingkun Xiao, Zihang Ye, et al. Swe-agi: Benchmarking specification-driven software construction with moonbit in the era of autonomous agents. *arXiv preprint arXiv:2602.09447*, 2026. URL https://arxiv.org/abs/2602.09447.

Jiale Zhao, Guoxin Chen, Fanzhe Meng, Minghao Li, Jie Chen, Hui Xu, Yongshuai Sun, Wayne Xin Zhao, Ruihua Song, Yuan Zhang, et al. Immersion in the github universe: Scaling coding agents to mastery. *arXiv preprint arXiv:2602.09892*, 2026. URL https://arxiv.org/abs/2602.09892.

Jiasheng Zheng, Boxi Cao, Boxi Yu, Yuzhong Zhang, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. Combinatorial synthesis: Scaling code rlvr via atomic decomposition and recombination. *arXiv preprint arXiv:2605.31058*, 2026. URL https://arxiv.org/abs/2605.31058.

Mingchen Zhuge, Changsheng Zhao, Dylan R. Ashley, Wenyi Wang, Dmitrii Khizbullin, Yunyang Xiong, Zechun Liu, Ernie Chang, Raghuraman Krishnamoorthi, Yuandong Tian, Yangyang Shi, Vikas Chandra, and Jürgen Schmidhuber. Agent-as-a-judge: Evaluate agents with agents. In *Proceedings of ICML*, volume 267, pp. 80569–80611, 13–19 Jul 2025. URL https://proceedings.mlr.press/v267/zhuge25a.html.

A Data Construction

A.1 Agents

The synthesis and review agents use MiMo-V2.5-Pro. Solver agents use MiMo-V2.5-Pro and MiMo-V2.5-Flash. These agents use Claude Code (Anthropic) as their harness.

B Experimental Settings

B.1 Reinforcement learning

We train the model using GRPO (Shao et al., 2024). Table 4 supplements the training setup in Section 3.2.

Table 4: Shared RL settings for both base models across CodeSpecWeave, SWE-PI, and MiMo Coding.

Setting	**Value**
Training batch size	128
PPO mini-batch size	128
Samples per task	16
Sampling temperature	1.0
Top-p	0.97
Prompt token limit	8,192
Response token limit	516,096

Training dynamics. Training score averages within-task pass rates over all tested groups before batch selection, excluding invalid outcomes and assigning zero to groups with fewer than two valid outcomes. Entropy is measured over masked training tokens; response length and main-agent turns are means per training context and sampled context, respectively. The sampled tasks change across updates.

B.2 Downstream evaluation

The evaluation uses three samples per task, a 512K context budget, temperature 1.0, top-$p = 0.95$.

C Specification Analysis

C.1 Categories and benchmark annotation

Table 5 summarizes the taxonomy across five primary categories, where functional semantics is further divided into three subcategories, yielding seven categories in total.

Table 5: Taxonomy and distribution of the 1,044 benchmark specifications. Functional specifications ($n = 470$) are partitioned into three fine-grained subcategories.

Specification Type	Scope	Count
Functional Semantics	End-to-end data transformation and algorithmic logic	470
Parsing & Representation	Syntax matching, encoding, and format serialization	182
Computation & Algorithms	Mathematical calculations and core algorithmic logic	145
Domain & Protocol Rules	Business constraints and protocol-specific semantics	143
Interface & I/O Contracts	Public APIs, argument types and defaults, input/output schemas	108
Control Flow & Ordering	Conditional branching, execution order, and concurrency	98
State & Side Effects	State transitions, atomicity, and resource lifecycles	181
Errors & Recovery	Input rejection, exception propagation, and fallback paths	187

We annotate the 1,044 specifications using a human-in-the-loop pipeline: MiMo-V2.5 (Xiaomi MiMo Team, 2026a) first proposes primary/secondary categories, auxiliary tags (compatibility, security, platform, observability), and rationales, which are subsequently audited by human annotators. In Figure 3, each specification is uniquely attributed to its primary category, while secondary categories track cross-cutting aspects.

Classification boundaries are grounded in the requirement's primary intent: interface contracts cover API signatures and schema constraints; functional semantics govern transformation logic; state obligations enforce invariants during failed transactions; and error handling captures explicit exceptions and retries. We further subdivide the 470 functional specifications into parsing, algorithmic computation, and domain protocols.

C.2 Comparison across datasets

Figure 1 compares our 100 benchmark requests against instances from four representative datasets: Scale-SWE-Verified (Zhao et al., 2026; Prime Intellect, 2026b), SWE-rebench-V2-Filtered-Verified (Badertdinov et al., 2026; Prime Intellect, 2026d), SWE-Lego-Real-Data-Verified (Tao et al., 2026; Prime Intellect, 2026c), and R2E-Gym-Subset-Verified (Jain et al., 2025; Prime Intellect, 2026a).

To quantify task complexity, we measure both specification volume and taxonomic coverage using an agent-based annotation pipeline. First, an LLM agent decomposes raw issue descriptions into atomic requirement clauses, extracting 836 benchmark clauses alongside our 1,044 ground-truth test specifications. Next, powered by MiMo-V2.5 (Xiaomi MiMo Team, 2026a), the agent categorizes each request across the taxonomy in Table 5.

D Case Study

D.1 Exception injection and fiber lifecycle

The mruby task adds `Fiber#raise` to inject an exception into a fiber suspended at `Fiber.yield`. For an unhandled exception, the target fiber must execute its `ensure` clauses, propagate the exception to the caller, and terminate.

All three solutions from the base MiMo-V2.5 model implement exception injection but leave the target fiber alive after an unhandled exception; one also skips the `ensure` clauses. All three solutions produced after CodeSpecWeave training handle cleanup and termination in the VM's exception-unwinding path and pass the tests.