SWE-CRAFTER: Scaling Executable Multilingual Software Engineering Data with Meta-Skill Agents

Shukai Liu[1] **Jian Yang**[1*] **Bo Jiang**[1*] **Junhang Cheng**[1]
Haowen Wang[2] **Xiao-Hui Li**[3] **Xianglong Liu**[1]

[1]SKLCCSE, Beihang University
[2]Beijing University of Posts and Telecommunications [3]Huawei
{skliu,jiayang,jiangbo}@buaa.edu.cn

Abstract

Software-engineering agents are increasingly expected to solve realistic repository-level tasks, but their training remains constrained by the limited availability of executable supervision beyond Python-centric repositories. Building such data across programming languages is difficult: repositories use different build systems, dependency managers, runtime assumptions, test interfaces, and failure modes, so separate language-specific pipelines are costly to reproduce and extend. We introduce SWE-CRAFTER, a unified agent-based framework for constructing executable multilingual SWE instances. SWE-CRAFTER keeps candidate mining, environment synthesis, fail-to-pass validation, and trajectory distillation in a shared pipeline, while language-specific construction knowledge is supplied through extensible skills induced from repository-level experience. These skills guide repository exploration and test-command synthesis while preserving repository-local evidence from documentation, manifests, CI files, and execution feedback as the final source of truth. Applying SWE-CRAFTER to eight programming languages yields 50,926 validated instances from 9,296 repositories and 68,851 distilled test-passing trajectories. Supervised fine-tuning on this resource produces SWE-CRAFTER-40B, which achieves 61.0% on SWE-bench Multilingual and 77.2% on SWE-bench Verified, showing that skill-guided construction provides scalable multilingual supervision for repository-level SWE agents.

1 Introduction

Software-engineering agents have advanced from standalone code generation to repository-level systems that resolve realistic development tasks [46, 41, 2]. Training and evaluating such agents requires executable SWE instances that link a natural-language intent, repository state, reference patch, and a test-based reward signal [22, 31]. Yet scalable construction outside Python-centric repositories remains difficult. Real projects vary in runtimes, package managers, native dependencies, CI conventions, and test interfaces, so a fixed environment recipe is brittle. At the same time, useful supervision requires fail-to-pass verification; hand-written language pipelines are costly to maintain, and ad hoc agent exploration repeats similar discovery work across repositories [31, 21, 48].

We introduce SWE-CRAFTER, a unified framework for multilingual SWE instance construction. SWE-CRAFTER keeps one agentic pipeline for candidate mining, environment synthesis, fail-to-pass validation, and trajectory distillation, but represents ecosystem-specific construction knowledge as configurable skills. Each skill guides repository inspection, dependency resolution, test invocation, and common failure recovery for a target language, allowing the same framework to extend across het-

*Co-corresponding authors.

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

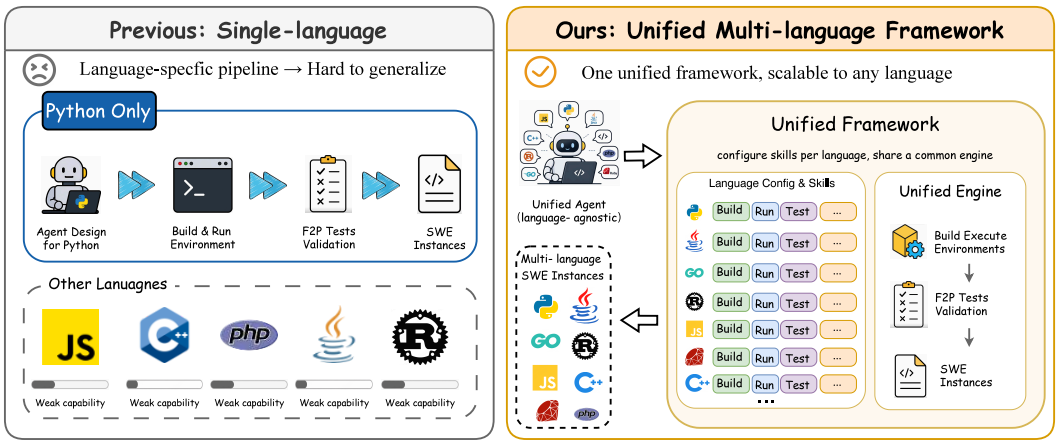


Figure 1: Motivation and overview of SWE-CRAFTER. Existing language-specific construction pipelines are difficult to extend across heterogeneous language ecosystems, while SWE-CRAFTER keeps a shared agent framework and factors ecosystem-specific knowledge into configurable skills.

erogeneous ecosystems. Figure 1 contrasts this design with language-specific construction pipelines and highlights the shared framework-plus-skill abstraction.

SWE-CRAFTER obtains skills through meta-skill induction: an agent explores seed repositories for a target language and summarizes recurring construction experience into a compact language-level skill. During large-scale construction, the skill serves as a prior, while repository-local evidence such as documentation, manifests, CI files, and test outputs remains the final source of truth. This design reduces repeated environment-discovery work without maintaining a separate hand-coded construction recipe for every language ecosystem.

Applying SWE-CRAFTER across eight language ecosystems, grouping C/C++ and JS/TS by shared build and repository conventions, yields a validated training resource with broad repository coverage and executable supervision. We use the resource to distill test-passing trajectories and train SWE-CRAFTER-4B, SWE-CRAFTER-9B, and SWE-CRAFTER-40B, with SWE-CRAFTER-40B reaching 61.0% on SWE-bench Multilingual and 77.2% on SWE-bench Verified. Beyond aggregate accuracy, we analyze how multilingual SWE data transfers across languages, scales with data size, changes trajectory statistics, and exposes language-specific construction difficulty.

Overall, this work makes three contributions: (1) SWE-CRAFTER, a meta-skill-driven unified agent framework for multilingual SWE instance construction; (2) a training resource containing 50,926 validated instances from 9,296 repositories and 68,851 distilled test-passing trajectories, used to train SWE-CRAFTER models; and (3) analyses of cross-language transfer, data scaling, trajectory complexity, and language-specific construction difficulty.

2 Method

2.1 Overview

SWE-CRAFTER is a unified agent-based framework for constructing multilingual software-engineering training data. As shown in Figure 2, the pipeline consists of three stages: raw instance collection, skills-based instance generation, and distillation. The first stage converts public GitHub development activity into raw task instances by mining high-quality repositories, collecting issue-related pull requests, and separating each candidate into problem evidence, a golden patch, a test patch, and version labels, following the common issue-to-task formulation introduced by repository-level SWE benchmarks [22, 31]. The second stage turns raw tasks into verified executable instances whose tests exhibit fail-to-pass behavior. It identifies the target language, retrieves the corresponding skill from a skills hub, and uses a construction agent to analyze the repository, resolve dependencies, write a Docker environment, and generate a test command, using an agentic coding interface such as Claude Code [2] in our implementation. The third stage runs a teacher model on verified instances and retains only trajectories whose submitted patches pass the generated tests.

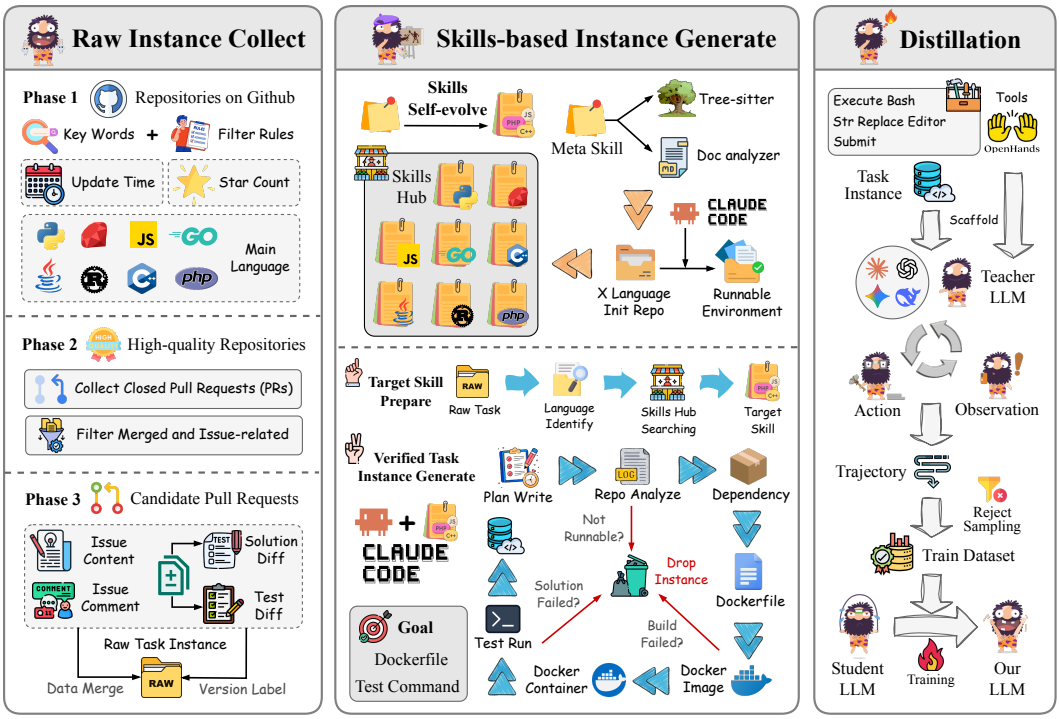


Figure 2: Overview of SWE-CRAFTER. **Raw Instance Collect**: mine issue-linked pull requests from high-quality GitHub repositories and convert them into raw task instances with problem evidence, patches, and version labels. **Skills-based Instance Generate**: retrieve language skills to guide agents in building executable environments, generating test commands, and retaining verified fail-to-pass instances. **Distillation**: run a teacher model on verified instances, keep only test-passing trajectories, and use them to train student software-engineering models.

This design keeps synthesis shared across languages while isolating ecosystem-specific knowledge in skills. Each instance follows a common format containing a repository, a base commit, a problem statement, a golden patch, a test patch, and an executable test interface. The same format can be consumed by environment builders, evaluation harnesses, and student-model training pipelines; skills only specialize language-dependent steps such as dependency management, build commands, test invocation, and failure recovery.

Formally, a raw task is represented as $r = (R, c_0, q, p_{\text{gold}}, p_{\text{test}}, \ell)$, where R is the repository, c_0 the base commit, q the problem statement, p_{gold} the golden patch, p_{test} the test patch, and ℓ the target language. Given a skills hub $\mathcal{S}$, construction is conditioned on the retrieved language skill:

$$s_\ell = \text{Retrieve}(\mathcal{S}, \ell), \qquad (D, u) = G_\theta(r, s_\ell), \tag{1}$$

where D is the generated execution environment and u is the test command. The instance verifier combines execution and semantic consistency gates,

$$\begin{aligned} v_{\text{exec}}(r, D, u) = \mathbb{I}\big[&\text{Exec}(R_{c_0} \oplus p_{\text{test}}, D, u) = \text{fail} \\ &\wedge \text{Exec}(R_{c_0} \oplus p_{\text{gold}} \oplus p_{\text{test}}, D, u) = \text{pass}\big], \\ v(r, D, u) = v_{\text{exec}}(r, D, u) &\wedge v_{\text{sem}}(q, p_{\text{test}}, u), \end{aligned} \tag{2}$$

where v_{sem} denotes the LLM-based semantic consistency judgment over the problem statement, test patch, and test command. Distillation then retains only teacher-model trajectories whose instance passes v and whose submitted patch $\hat{p}_\tau$ passes the generated test command:

$$\mathcal{D}_{\text{sft}} = \{(r, \tau) : v(r, D, u) = 1,\ \text{Exec}(R_{c_0} \oplus \hat{p}_\tau \oplus p_{\text{test}}, D, u) = \text{pass}\}. \tag{3}$$

2.2 Raw Instance Collection

The left panel of Figure 2 shows the raw instance collection stage. SWE-CRAFTER first discovers candidate repositories from GitHub using language-aware keywords and lightweight quality filters.

We filter repositories by metadata such as update time, star count, dominant programming language, and permissive licenses, as detailed in Appendix I. To reduce evaluation contamination, this repository-level filter also removes any repository that appears in the benchmark task sets used for evaluation in Section 3, including SWE-bench Multilingual and SWE-bench Verified. We then collect closed pull requests from the retained repositories. To improve the likelihood that a change corresponds to a well-specified software-engineering task, we keep merged pull requests that are linked to issue discussions and contain both implementation changes and test-relevant changes.

For each candidate pull request, the pipeline recovers the issue content, discussion comments, golden patch, and test patch. The issue text and comments form the natural-language problem evidence, while the golden and test patches provide the executable supervision needed for later validation. We also attach version labels, including the repository identifier and base commit, so that the task can be reconstructed deterministically. The output of this stage is a raw task instance that is language-agnostic in format but still preserves the repository evidence needed for language-specific construction.

2.3 Skills Hub and Meta-Skill Evolution

The middle panel of Figure 2 illustrates how SWE-CRAFTER factors language knowledge into skills. Instead of maintaining a separate hand-written construction pipeline for each language, we maintain a skills hub containing language-specific skills and reusable meta-skills for repository analysis. Before large-scale construction, SWE-CRAFTER asks an agent to solve a small set of seed repositories for each language, typically around 50 repositories. During this self-evolution process, the agent observes recurring repository structures, package managers, runtime constraints, build tools, CI conventions, test-selection patterns, and common failure modes.

The agent summarizes these experiences into a compact language skill. A skill does not prescribe a single environment recipe. It records construction priors: which manifest files to inspect, how to infer runtime versions, which package managers and build tools are common, how tests are usually invoked, and which failures often require additional system packages or services. During instance generation, SWE-CRAFTER identifies the language of a raw task, searches the skills hub, and retrieves a target skill. The skill guides repository exploration and environment synthesis, but concrete repository evidence remains the final source of truth.

2.4 Skills-Based Instance Generation

Given a raw task and its target skill, SWE-CRAFTER uses a construction agent to produce a verified executable instance. The agent first writes a construction plan, then analyzes the repository documentation, manifests, lock files, CI configuration, helper scripts, build metadata, and test directories. Based on this analysis, the agent infers dependencies, writes a Dockerfile, builds a Docker image, starts a container, and runs the target test command.

Validation is interleaved with construction. If the repository cannot be made runnable, the Docker image cannot be built, the tests are unstable, or the golden patch does not pass under the generated command, the candidate is discarded after the available repair budget is exhausted. Otherwise, SWE-CRAFTER applies an explicit fail-to-pass check: at least one target test must fail on the base commit and pass after the golden patch is applied, defining v_{exec} in Equation 2. We additionally use an LLM judge as a semantic consistency check over the problem statement, the unit tests introduced by the test patch, and the generated test command. Candidates are retained only when the judge finds that the tests and command exercise behavior aligned with the issue description, defining v_{sem} and reducing semantic mismatch in the constructed data. The exported verified task instance contains the problem statement, base commit, golden patch, test patch, Dockerfile, and test command. This verifier is shared across all languages, while the steps used to reach a runnable environment remain skill-conditioned.

2.5 Trajectory Distillation

The right panel of Figure 2 shows the distillation stage. We use a rejection-sampling style procedure to turn verified instances into supervised training data. First, we run a teacher model on the verified task instances and record the full trajectory corresponding to each run. We then evaluate the submitted patch with the generated test command and mark the run as resolved only when the patch passes the same target tests used for construction.

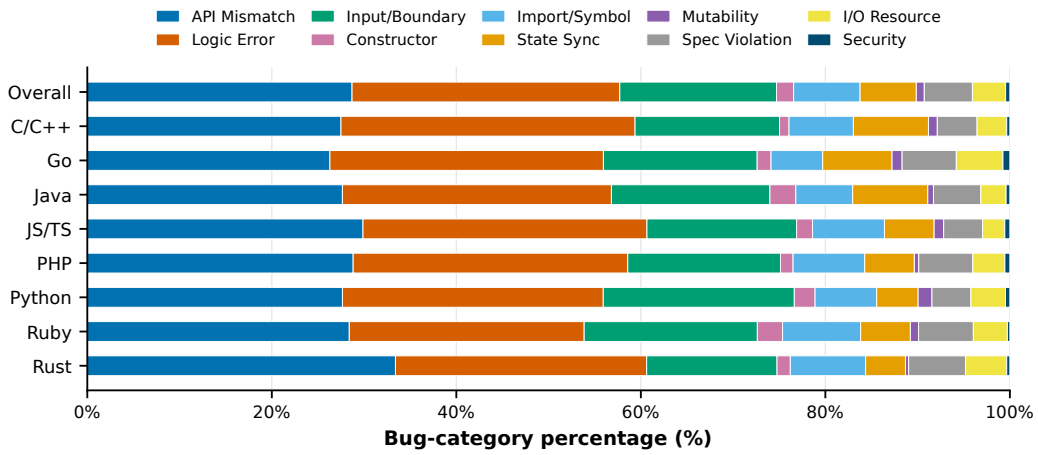


Figure 3: Bug-category distribution of SWE-CRAFTER instances. Each row shows the normalized category distribution for the constructed instances.

Only trajectories corresponding to resolved instances are retained for fine-tuning. Failed rollouts are rejected rather than used as positive supervision. The resulting distillation set therefore contains complete demonstrations that end in test-passing patches, allowing student models to learn from the repository exploration, debugging decisions, and edit sequence that led to a valid solution rather than treating the final patch alone as sufficient supervision.

2.6 Dataset Statistics

Table 1a compares SWE-CRAFTER with representative SWE training datasets by executable instance count, data source, repository diversity, trajectory count, and multilingual coverage [21, 31, 48, 39, 7]. SWE-CRAFTER covers more repositories than prior Python-centric resources while retaining executable validation, multilingual coverage, and distillation trajectories. Table 1b characterizes golden-patch structure: the median instance modifies two files and 30 total lines, while the 95th percentile touches 14 files and changes more than 550 lines.

Table 1: Dataset-scale comparison and patch-level statistics. Counts in (a) follow the corresponding papers or public releases; Inst. denotes executable instances with validated tests, Src. the primary data source, ML multilingual coverage, and N/A no comparable trajectory set. In (b), Edit is the hunk-level minimum of added and deleted lines, and Total is added plus deleted lines.

(a) Existing SWE training datasets.

Dataset	Inst.	Src.	Repo	Traj.	ML
R2E-Gym	8.1k	Synth.	13	3.2k	✗
SWE-Gym	2.4k	Real	11	491	✗
SWE-smith	50.0k	Synth.	128	5.0k	✗
SWE-Mirror	60.7k	Synth.	40	6.4k	✓
SWE-rebench-v2	32.1k	Real	3.6k	N/A	✓
SWE-CRAFTER	**50.9k**	Real	**9.3k**	**68.9k**	✓

(b) SWE-CRAFTER patch statistics.

Stat.	Files	Del.	Add	Edit	Total
Mean	4.3	49.7	119.1	22.5	168.8
Std.	5.8	54.1	113.6	30.5	147.8
P25	1.0	1.0	6.0	1.0	9.0
P50	2.0	5.0	22.0	3.0	30.0
P75	4.0	19.0	71.0	13.0	98.0
P95	14.0	145.0	399.0	78.0	552.8

Figure 3 examines semantic coverage using the BugPilot taxonomy [34]. We label the constructed instances with MiniMax M2.5 and normalize the bug-category distribution. The largest shares come from API mismatches, logic errors, and input-boundary cases across languages.

3 Experiments

3.1 Experimental Setup

We evaluate on two repository-level benchmarks: SWE-bench Multilingual [48], which contains 300 tasks spanning nine programming languages and 42 repositories, and SWE-bench Verified [22], a human-filtered 500-instance subset of SWE-bench. We report single-attempt resolved rates, counting an instance as resolved only when the submitted patch passes the benchmark tests. Our training data uses eight language ecosystems rather than the benchmark's nine-language taxonomy, grouping C/C++ and JS/TS by shared build, repository, and target-test conventions.

For training, validated SWE-CRAFTER instances serve as executable distillation environments. We run OpenHands [41] with MiniMax M2.5 [26] as the teacher model and retain only reward $= 1$ trajectories whose submitted patches pass the repository unit tests, yielding 68,851 test-passing trajectories. We fine-tune Qwen3.5-4B-Base, Qwen3.5-9B-Base [32], and IQuest-Coder-V1-40B-Instruct [44] into SWE-CRAFTER-4B, SWE-CRAFTER-9B, and SWE-CRAFTER-40B with MS-Swift [28]. Unless otherwise stated, all checkpoints are evaluated with OpenHands [41] under the same harness and without inference-time scaling.

3.2 Baselines and Main Results

Table 2: Main evaluation results on SWE-bench Multilingual and SWE-bench Verified. Scores are resolved rates (%); lock icons mark closed systems.

Method			Resolved rate (%)	
Model	**Size**	**Scaffold**	**SWE-Multi**	**SWE-Verified**
Frontier closed models				
Gemini 3 Flash [17]	🔒	mini-SWE-agent	72.7	75.8
Claude 4.6 Opus [5]	🔒	mini-SWE-agent	72.0	75.6
Claude Opus 4.5 [3]	🔒	OpenHands	75.2	79.0
Claude Sonnet 4.5 [4]	🔒	OpenHands	66.0	74.6
GPT-5.2 [30]	🔒	mini-SWE-agent	66.7	72.8
GPT-5 Mini [29]	🔒	iFlow CLI	49.7	59.3
Open-weight coding models				
Kimi K2 [38]	1043B-A32B	mini-SWE-agent	48.7	64.8
Kimi K2.5 [37]	1000B-A32B	mini-SWE-agent	67.3	70.8
GLM-5 [51]	744B-A40B	mini-SWE-agent	69.7	72.8
DeepSeek V3.2 [23]	671B-A37B	OpenHands	61.8	72.6
Qwen3-Coder 480B [8]	480B-A35B	OpenHands	54.7	69.6
GLM-4.7 [52]	358B-A32B	OpenHands	66.7	73.8
GLM-4.5 [52]	358B-A32B	OpenHands	53.8	68.0
MiniMax M2.5 [26]	229B-A10B	mini-SWE-agent	68.3	75.8
Nemotron 3 Super [10]	120B-A12B	OpenHands	45.8	60.5
GPT-OSS 120B [1]	120B-A20B	mini-SWE-agent	34.8	43.9
GLM-4.5 Air [52]	106B-A12B	iFlow CLI	38.2	56.2
Qwen3-Coder-Next [8]	80B-A3B	OpenHands	64.3	71.3
ROME [40]	30B-A3B	iFlow CLI	40.0	57.4
Qwen3-Coder 30B [8]	30B-A3B	OpenHands	34.7	51.6
Gemma4 26B [18]	26B-A4B	OpenHands	17.3	17.4
IQuest-Coder V1 Inst. [44]	40B	OpenHands	30.0	76.2
Gemma4 31B [18]	31B	OpenHands	51.7	52.0
Devstral Small 2 [27]	24B	iFlow CLI	27.0	51.9
SWE-CRAFTER models				
SWE-CRAFTER-4B	4B	OpenHands	**50.3**	**58.6**
SWE-CRAFTER-9B	9B	OpenHands	**56.7**	**65.6**
SWE-CRAFTER-40B	40B	OpenHands	**61.0**	**77.2**

Table 2 compares frontier closed systems, open-weight coding models, and our SWE-CRAFTER checkpoints, restricted to systems with SWE-bench Multilingual results. Within the SWE-CRAFTER family, performance improves consistently with model scale: SWE-CRAFTER-4B reaches 50.3% on SWE-bench Multilingual and 58.6% on SWE-bench Verified, SWE-CRAFTER-9B improves to

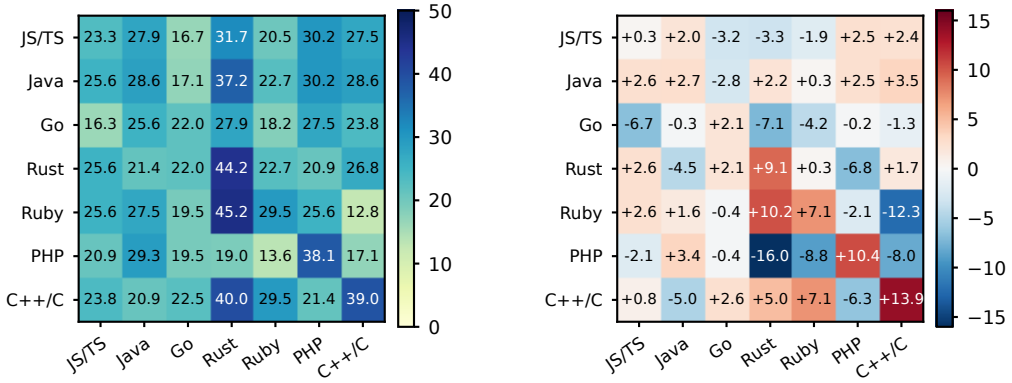
56.7% and 65.6%, and SWE-CRAFTER-40B further reaches 61.0% and 77.2%. This monotonic trend shows that the distilled multilingual trajectories are useful across different student sizes and base-model families rather than being tied to a single checkpoint.

The strongest SWE-CRAFTER model is especially competitive on SWE-bench Verified, where SWE-CRAFTER-40B approaches the strongest frontier closed systems in the table while remaining a 40B open-weight checkpoint. On SWE-bench Multilingual, the 4B and 9B variants already exceed several much larger open-weight baselines, and the 40B model provides the best result among our trained checkpoints. Together, these results support the central claim that validated multilingual SWE instances and test-passing trajectories provide effective post-training supervision for repository-level software agents.

4 Analysis

4.1 Cross-Language Transfer Analysis

We analyze cross-language transfer by fine-tuning IQuest-Coder-V1-14B-Base on 3K single-language examples with matched training steps. Figure 4 reports absolute pass rates and column-normalized transfer gains, where positive entries indicate above-average performance for the same target language. Rust is the easiest evaluation target on average and Go is the hardest, while C++/C and Java provide the strongest average transfer. The normalized view further shows that transfer is asymmetric: the largest gains are often in-language or near-language, but Ruby-trained models also improve Rust relative to the per-target average. Conversely, several off-diagonal entries are negative, showing that single-language training can hurt some targets. These patterns indicate that language-specific supervision induces structured rather than uniform capability shifts across programming languages. This motivates multilingual training sets that preserve language identity rather than mixing all examples as interchangeable code-editing traces. It also explains why we separate the absolute heatmap from the normalized heatmap: the former captures target difficulty, while the latter isolates whether a source language provides extra benefit beyond that target's average difficulty. The absence of a single dominant source language further supports training on a balanced multilingual mixture instead of relying on one high-resource ecosystem.



(a) Pass rates (%). (b) Transfer gains (pp).

Figure 4: Cross-language transfer under single-language fine-tuning.

4.2 Training Data Scaling Analysis

We next scale IQuest-Coder-V1-14B-Base from 3K to 45K multilingual training examples and evaluate each checkpoint with the same inference harness. Figure 5 reports pass rate, average token cost, and resolved tasks per billion tokens. More data improves both accuracy and efficiency: SWE-Multi rises from 18.8% to 38.0%, and SWE-Verified rises from 37.0% to 54.0%. The trend is not perfectly monotonic, with SWE-Multi dipping from 8K to 16K, but the overall scaling direction is positive. At the same time, token cost falls from 10.2M to 7.4M and from 8.2M to 5.6M tokens

per task, respectively. Token-normalized efficiency therefore improves from 18.5 to 51.5 on SWE-Multi and from 45.2 to 96.4 on SWE-Verified, showing that larger training sets also make inference trajectories more productive. The efficiency trend is important for SWE agents because pass rate alone can hide increasingly expensive search behavior; here, additional training data improves the chance of solving a task while reducing the average amount of interaction required to reach a successful patch. This suggests that the added examples do not merely encourage longer search, but improve the model's ability to localize relevant files, select tests, and stop after a sufficient patch. In practice, this makes larger multilingual supervision useful both for accuracy and for inference-time efficiency. The 16K dip also suggests that intermediate checkpoints can be noisy, so we treat the scaling curve as an aggregate trend rather than claiming strict monotonic improvement at every data size.

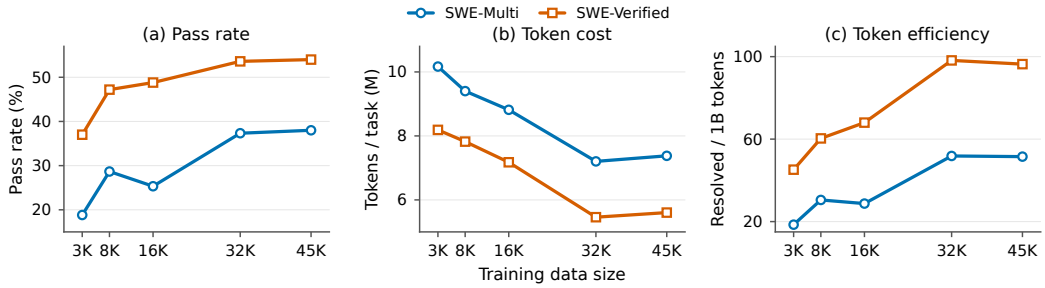


Figure 5: Training-data scaling improves pass rate and token-normalized solve efficiency.

4.3 Trajectory Statistics of Distillation Data

We estimate trajectory length from message content, reasoning content, and serialized tool calls, and count assistant tool-call turns excluding the final submission. Figure 6 shows that our trajectories are longer than SWE-Smith and close to SWE-rebench-v2: the median uses 43.9K tokens and 60 tool-call turns, compared with 34.4K/43 for SWE-Smith and 42.0K/58 for SWE-rebench-v2. The data therefore captures more repository exploration, execution feedback, and iterative debugging than shorter single-file-oriented traces. Such trajectories provide denser supervision for when to gather evidence, execute tests, refine patches, and stop editing, which are common failure modes in long-horizon repository-level agents. The comparison also separates trajectory richness from raw dataset size: our traces preserve long-horizon exploration and test-feedback decisions while being collected from verified training instances rather than benchmark-style evaluation tasks.



Figure 6: Distributions of trajectory token counts and tool-call turns.

4.4 Language-wise Construction Difficulty

We measure language-wise construction difficulty by sampling eight teacher-model rollouts per verified instance. Reward-1 yield is the fraction with at least one test-passing rollout; trajectory complexity summarizes token count, tool-call turns, and edited files. Figure 7 shows that JS/TS, Ruby, and PHP have the highest yields, while Go, Java, and Rust are harder to distill. C/C++ and Rust require the longest trajectories. These patterns show that language ecosystems impose distinct

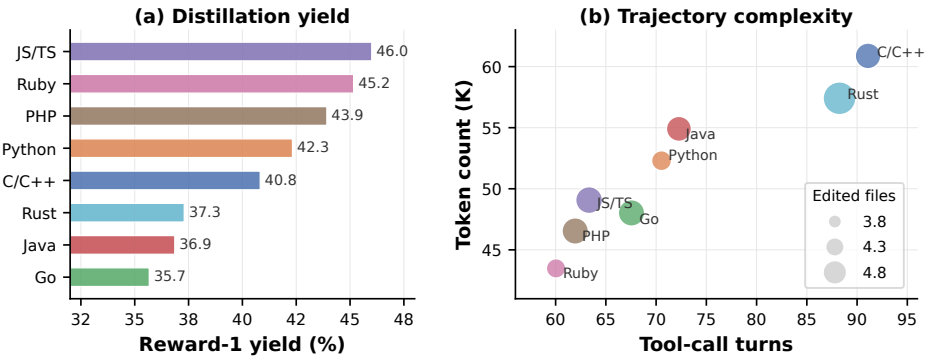


Figure 7: Language-wise construction difficulty. Left: reward-1 yield under eight teacher-model rollouts; right: average trajectory complexity.

build, dependency, and test-execution costs, supporting a shared pipeline with language-dependent skills and identifying where better verifier coverage is most valuable. The two panels should be read jointly. High-yield languages such as JS/TS and Ruby provide efficient sources of successful trajectories, whereas C/C++ and Rust combine moderate yield with expensive trajectories, reflecting native toolchains, compilation latency, and dependency constraints. This explains why a single generic construction recipe is brittle: the same agent loop must preserve repository-local evidence, but the search priorities and repair heuristics need to adapt to each ecosystem.

5 Related Work

Benchmarks for Software Tasks Repository-level SWE evaluation has moved from standalone code generation to executable issue resolution. SWE-bench introduced real GitHub issues with repository tests, and SWE-bench Verified provides a human-filtered subset for more reliable measurement [22]. Recent benchmarks extend this setting to multilingual, live, decontaminated, visual, long-horizon, terminal, and project-level tasks [48, 50, 6, 54, 47, 14, 25, 56]. These benchmarks require agents to edit interacting files, execute tests, and recover from failures in realistic repositories.

Data Construction and Post-Training Training repository-level agents further requires executable environments and verified trajectories. SWE-agent and OpenHands provide repository-facing loops for inspection, shell execution, editing, and test validation [46, 41]. Data-construction efforts build real, synthetic, mirrored, language-agnostic, and automated SWE resources [31, 21, 48, 39, 7, 19]. These systems establish executable validation as the core supervision signal, but construction remains difficult across heterogeneous language ecosystems. Complementary work studies SFT, RL, agent-native mid-training, skill priors, context management, and bug generation [13, 35, 55, 33, 36, 53, 16, 9, 49, 24, 34]. SWE-CRAFTER follows this data-centric direction but centers on reusable multilingual construction and distilled trajectories.

6 Conclusion

We presented SWE-CRAFTER, a meta-skill-driven unified agent framework for constructing executable multilingual SWE instances. SWE-CRAFTER keeps a shared agent-based workflow and adapts it with language-level skills induced from seed repositories, preserving repository-local evidence while making environment synthesis, target-test selection, and repair heuristics reusable across heterogeneous language ecosystems.

Using this framework, we construct 50,926 validated instances and 68,851 test-passing trajectories across eight programming languages. Supervised fine-tuning on these trajectories yields SWE-CRAFTER-40B, which achieves 61.0% on SWE-bench Multilingual and 77.2% on SWE-bench Verified. The analyses show that multilingual SWE supervision transfers asymmetrically across languages, benefits from scale, and exposes different construction costs across ecosystems. These findings suggest that scalable SWE post-training requires not only more issue-resolution data, but also executable validation and reusable construction knowledge specialized to each language ecosystem.

Acknowledgments and Disclosure of Funding

This work was supported by the National Natural Science Foundation of China (Grant No. 62525601), the Fundamental Research Funds for the Central Universities (Grant No. GW2025-19), and the State Key Laboratory of Complex & Critical Software Environment (SKLCCSE) (Grant No. SKLCCSE-2025ZX-26). The authors declare no competing interests.

References

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

[2] Anthropic. Claude code. `https://claude.com/product/claude-code`, 2025. Accessed: 2026-05-06.

[3] Anthropic. Claude model system card. Claude Opus 4.5, 2025. Accessed: 2026-05-06.

[4] Anthropic. Claude model system card. Claude Sonnet 4.5, 2025. Accessed: 2026-05-06.

[5] Anthropic. Claude model system card. Claude Opus 4.6, 2026. Accessed: 2026-05-06.

[6] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

[7] Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, and Alexander Golubev. Swe-rebench v2: Language-agnostic swe task collection at scale. *arXiv preprint arXiv:2602.23866*, 2026.

[8] Ruisheng Cao, Mouxiang Chen, Jiawei Chen, Zeyu Cui, Yunlong Feng, Binyuan Hui, Yuheng Jing, Kaixin Li, Mingze Li, Junyang Lin, Zeyao Ma, Kashun Shum, Xuwu Wang, Jinxi Wei, Jiaxi Yang, Jiajun Zhang, Lei Zhang, Zongmeng Zhang, Wenting Zhao, and Fan Zhou. Qwen3-coder-next technical report. *arXiv preprint arXiv:2603.00729*, 2026.

[9] Shiyi Cao, Dacheng Li, Fangzhou Zhao, Shuo Yuan, Sumanth R Hegde, Connor Chen, Charlie Ruan, Tyler Griggs, Shu Liu, Eric Tang, et al. Skyrl-agent: Efficient rl training for multi-turn llm agent. *arXiv preprint arXiv:2511.16108*, 2025.

[10] Aakshita Chandiramani, Aaron Blakeman, Abdullahi Olaoye, Abhibha Gupta, Abhilash Somasamudramath, Abhinav Khattar, Adeola Adesoba, Adi Renduchintala, Adil Asif, Aditya Agrawal, et al. Nemotron 3 super: Open, efficient mixture-of-experts hybrid mamba-transformer model for agentic reasoning. *arXiv preprint arXiv:2604.12374*, 2026.

[11] Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[13] Mouxiang Chen, Lei Zhang, Yunlong Feng, Xuwu Wang, Wenting Zhao, Ruisheng Cao, Jiaxi Yang, Jiawei Chen, Mingze Li, Zeyao Ma, et al. Swe-universe: Scale real-world verifiable environments to millions. *arXiv preprint arXiv:2602.02361*, 2026.

[14] Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

[15] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

[16] Alexander Golubev, Maria Trofimova, Sergei Polezhaev, Ibragim Badertdinov, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Sergey Abramov, Andrei Andriushchenko, Filipp Fisin, et al. Training long-context, multi-turn software engineering agents with reinforcement learning. *arXiv preprint arXiv:2508.03501*, 2025.

[17] Google. A new era of intelligence with gemini 3. `https://blog.google/products-and-platforms/products/gemini/gemini-3`, 2025. Accessed: 2026-05-06.

[18] Google DeepMind. Our most intelligent open models, built from gemini 3 research and technology to maximize intelligence-per-parameter. `https://deepmind.google/models/gemma/gemma-4/`, 2026. Accessed: 2026-05-06.

[19] Lianghong Guo, Yanlin Wang, Caihua Li, Wei Tao, Pengyu Yang, Jiachi Chen, Haoyu Song, Duyu Tang, and Zibin Zheng. Swe-factory: Your automated factory for issue resolution training data and evaluation benchmarks. *arXiv preprint arXiv:2506.10954*, 2025.

[20] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[21] 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.

[22] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

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

[24] Shukai Liu, Jian Yang, Bo Jiang, Yizhi Li, Jinyang Guo, Xianglong Liu, and Bryan Dai. Context as a tool: Context management for long-horizon swe-agents. *arXiv preprint arXiv:2512.22087*, 2025.

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

[26] MiniMax. Minimax-m2.5. `https://huggingface.co/MiniMaxAI/MiniMax-M2.5`, 2026. Accessed: 2026-05-06.

[27] Mistral AI. Introducing: Devstral 2 and mistral vibe cli. | mistral ai. `https://mistral.ai/news/devstral-2-vibe-cli`, 2025. Accessed: 2026-05-06.

[28] ModelScope. MS-Swift: Scalable lightweight infrastructure for fine-tuning. `https://github.com/modelscope/ms-swift`, 2024. Accessed: 2026-05-07.

[29] OpenAI. Introducing GPT-5. `https://openai.com/index/introducing-gpt-5/`, 2025. Accessed: 2026-05-06.

[30] OpenAI. Introducing GPT-5.2. `https://openai.com/index/introducing-gpt-5-2/`, 2025. Accessed: 2026-05-06.

[31] Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym. In *Proceedings of the 42nd International Conference on Machine Learning (ICML 2025)*, 2025. arXiv:2412.21139, accepted at ICML 2025.

[32] Qwen Team. Qwen3.5: Towards native multimodal agents. `https://qwen.ai/blog?id=qwen3.5`, 2026. Accessed: 2026-05-06.

[33] Huatong Song, Lisheng Huang, Shuang Sun, Jinhao Jiang, Ran Le, Daixuan Cheng, Guoxin Chen, Yiwen Hu, Zongchao Chen, Yiming Jia, et al. Swe-master: Unleashing the potential of software engineering agents via post-training. *arXiv preprint arXiv:2602.03411*, 2026.

[34] Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, et al. Bugpilot: Complex bug generation for efficient learning of swe skills. *arXiv preprint arXiv:2510.19898*, 2025.

[35] Shuang Sun, Huatong Song, Lisheng Huang, Jinhao Jiang, Ran Le, Zhihao Lv, Zongchao Chen, Yiwen Hu, Wenyang Luo, Wayne Xin Zhao, et al. Swe-world: Building software engineering agents in docker-free environments. *arXiv preprint arXiv:2602.03419*, 2026.

[36] 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.

[37] Kimi Team, Tongtong Bai, Yifan Bai, Yiping Bao, SH Cai, Yuan Cao, Y Charles, HS Che, Cheng Chen, Guanduo Chen, et al. Kimi k2.5: Visual agentic intelligence. *arXiv preprint arXiv:2602.02276*, 2026.

[38] Kimi Team, Yifan Bai, Yiping Bao, Y Charles, Cheng Chen, Guanduo Chen, Haiting Chen, Huarong Chen, Jiahao Chen, Ningxin Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

[39] 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.

[40] Weixun Wang, XiaoXiao Xu, Wanhe An, Fangwen Dai, et al. Let it flow: Agentic crafting on rock and roll - building the rome model within an open agentic learning ecosystem. *arXiv preprint arXiv:2512.24873*, 2025.

[41] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

[42] Jiajun Wu, Jian Yang, Wei Zhang, Lin Jing, Yuqing Ma, Ensheng Shi, Yuchi Ma, Zhoujun Li, and Xianglong Liu. Ucoder: Unsupervised code generation by internal probing of large language models. *arXiv preprint arXiv:2512.17385*, 2025.

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

[44] Jian Yang, Wei Zhang, Shawn Guo, Zhengmao Ye, Lin Jing, Shark Liu, Yizhi Li, Jiajun Wu, Cening Liu, X Ma, et al. Iquest-coder-v1 technical report. *arXiv preprint arXiv:2603.16733*, 2026.

[45] Jian Yang, Wei Zhang, Jiajun Wu, Junhang Cheng, Shawn Guo, Haowen Wang, Weicheng Gu, Yaxin Du, Joseph Li, Fanglin Xu, et al. Incoder-32b: Code foundation model for industrial scenarios. *arXiv preprint arXiv:2603.16790*, 2026.

[46] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[47] John Yang, Carlos E Jimenez, Alex L Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R Narasimhan, et al. Swe-bench multimodal: Do ai systems generalize to visual software domains? *arXiv preprint arXiv:2410.03859*, 2024.

[48] 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. *arXiv preprint arXiv:2504.21798*, 2025.

[49] Zonghan Yang, Shengjie Wang, Kelin Fu, Wenyang He, Weimin Xiong, Yibo Liu, Yibo Miao, Bofei Gao, Yejie Wang, Yingwei Ma, et al. Kimi-dev: Agentless training as skill prior for swe-agents. *arXiv preprint arXiv:2509.23045*, 2025.

[50] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

[51] Aohan Zeng, Xin Lv, Zhenyu Hou, Zhengxiao Du, Qinkai Zheng, Bin Chen, Da Yin, Chendi Ge, Chenghua Huang, Chengxing Xie, et al. Glm-5: from vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026.

[52] Aohan Zeng, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, Chengxing Xie, Cunxiang Wang, Da Yin, Hao Zeng, Jiajie Zhang, et al. Glm-4.5: Agentic, reasoning, and coding (arc) foundation models. *arXiv preprint arXiv:2508.06471*, 2025.

[53] Ji Zeng, Dayuan Fu, Tiantian Mi, Yumin Zhuang, Yaxing Huang, Xuefeng Li, Lyumanshan Ye, Muhang Xie, Qishuo Hua, Zhen Huang, et al. davinci-dev: Agent-native mid-training for software engineering. *arXiv preprint arXiv:2601.18418*, 2026.

[54] 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! *arXiv preprint arXiv:2505.23419*, 2025.

[55] 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.

[56] Qianhui Zhao, Li Zhang, Fang Liu, Junhang Cheng, Chengru Wu, Junchen Ai, Qiaoyuanhe Meng, Lichen Zhang, Xiaoli Lian, Shubin Song, et al. Towards realistic project-level code generation via multi-agent collaboration and semantic architecture modeling. *arXiv preprint arXiv:2511.03404*, 2025.

A Additional Related Work

Benchmarks for Software Tasks Code models have moved quickly from function-level generation into repository work, helped by stronger model families and industrial code models [32, 51, 23, 45]. Evaluation has followed: benchmarks now range from standalone programming problems [12, 20, 15] to executable software-engineering tasks. SWE-bench introduced real GitHub issue resolution with repository tests, and SWE-bench Verified provides a human-filtered subset [22]. This changed the evaluation target from producing a plausible patch to producing a patch that survives the repository's own verifier. The repository-test setting has since expanded in several directions: multilingual repositories [48, 50], visual software domains [47], long-horizon tasks [14], live or decontaminated evaluation [54, 6], command-line tasks [25], and project-level software development [56].

Methods for Software Tasks Work on software-task methods has grown around agent interfaces, training environments, and model specialization. SWE-agent and OpenHands define repository-facing agent loops for issue resolution [46, 41], while Claude Code and iFlow package similar capabilities into CLI tools [2, 40]. Data-centric work builds the supervision and runtime layer for these agents: executable environments and verifiers, synthetic SWE data, mirrored issues, and language-agnostic task collections [31, 21, 48, 39, 7]. SWE-Factory is especially related because it automates issue-resolution data construction with LLM-based multi-agent environment construction and automatic Fail-to-Pass validation [19]. SWE-CRAFTER is complementary: it introduces meta-skill induction so that a shared construction pipeline can be adapted across more language ecosystems, and it studies large-scale multilingual trajectory distillation for post-training. Other efforts scale verifiable SWE environments across many repositories [13, 35, 55]. Post-training work adapts the models themselves

through supervised fine-tuning, agent-native mid-training, reinforcement learning, skill priors, context management, and bug generation [33, 36, 53, 16, 9, 49, 24, 34]. These methods target different failure modes, from weak repository navigation to brittle long-context interaction and sparse executable feedback. Related code-agent studies cover language-agent fine-tuning, unsupervised code generation, and large-scale terminal trajectories [11, 42, 43].

B Data Availability

We release the SWE-CRAFTER dataset on Hugging Face at `https://huggingface.co/datasets/skLiu/SWE-Crafter`. The released data support research on executable multilingual software-engineering tasks. Each task follows the common format described in Section 2, including the upstream repository, base commit, problem statement, golden patch, test patch, Dockerfile, and test command. These records support inspection of task supervision, reconstruction of repository states, and task-level execution checks. The accompanying dataset documentation describes the record format, intended use, provenance, and upstream-license considerations.

C Data-Based Reproducibility

The released task records provide a data-based route to verify the executable supervision used by SWE-CRAFTER. For each instance, a researcher can retrieve the upstream repository at the recorded base commit, apply the test patch, and use the Dockerfile and test command to reconstruct the task environment. Running the target tests before and after applying the golden patch checks the Fail-to-Pass execution criterion in Equation 2: the tests fail on the base state and pass on the patched state. The problem statement, patches, and test command also expose the evidence needed to inspect whether the supervision matches the requested behavior.

Section 2 describes the construction and filtering procedure, and Appendix F documents the language skills used for environment construction. For the model experiments, Section 3 and Appendix H report the teacher model, test-passing trajectory-selection criterion, student model families, supervised fine-tuning recipe, and single-attempt benchmark protocol. OpenHands [41] and MS-Swift [28] provide the public agent and training frameworks used in these experiments. Together, the task data, explicit validation procedure, and reported experimental settings support independent task verification and reuse in software-engineering model training and evaluation.

D Comparison with Automated SWE Construction Pipelines

Table 3 compares SWE-CRAFTER with representative SWE training-data and task-construction pipelines [31, 21, 48, 39, 7, 19]. SWE-Factory directly addresses the automation problem by constructing Docker-based issue-resolution environments and applying automatic Fail-to-Pass validation. SWE-CRAFTER builds on the same broad goal of executable SWE data construction, but targets a different bottleneck: making a shared agent-based pipeline easier to extend across heterogeneous programming-language ecosystems. It does so through meta-skill induction and uses the validated multilingual instances to distill a substantially larger set of test-passing trajectories for student-model training.

Table 3: Comparison with automated SWE data-construction pipelines. Counts follow the corresponding papers or public releases. Inst. denotes validated executable instances or the reported construction-valid subset; Traj. denotes released or used training trajectories when available; Skill marks language-level skill induction.

Work	Lang.	Inst.	Traj.	Skill	Training use
SWE-Gym	1	2.4k	491	✗	Python SWE SFT
R2E-Gym	1	8.1k	3.2k	✗	Python agent training
SWE-smith	1	50.0k	5.0k	✗	Python SWE SFT
SWE-Mirror	Multi	60.7k	6.4k	✗	Multilingual SFT
SWE-rebench-v2	Multi	32.1k	N/A	✗	Evaluation collection
SWE-Factory	4	337/671	2.8k Py	✗	Exploratory Python SFT
SWE-CRAFTER	8	**50.9k**	**68.9k**	✓	Multilingual distillation

This comparison clarifies the originality boundary of SWE-CRAFTER. SWE-Factory shows that automated environment construction and Fail-to-Pass validation are feasible for issue-resolution data, and reports an exploratory model-training study on Python task instances. SWE-CRAFTER instead focuses on how to make the construction process language-extensible: language skills are induced from seed repositories and then condition the shared construction agent without replacing repository-local evidence. This design supports eight language ecosystems and produces 68.9k test-passing trajectories, enabling the multilingual post-training experiments reported in Section 3.

E Language Skill Ablation

To directly assess whether language skills are useful during instance construction, we run a controlled ablation on a fixed subset of instances using Qwen3-Coder-30B-A3B [8] as the construction model. The two agents are evaluated on the same candidate set within each language and share the same Stage-2 construction pipeline and execution budget. The no-skill agent receives only the generic construction instruction, while the skill agent additionally receives the corresponding language skill. We report the instance-level construction success rate:

$$\mathrm{SuccessRate}_\ell(a) = \frac{1}{|\mathcal{D}_\ell|} \sum_{x \in \mathcal{D}_\ell} \mathbf{1}\{\mathrm{Verify}(a, x) = 1\},$$

where $\mathcal{D}_\ell$ is the fixed candidate set for language ℓ and a denotes the construction agent. Table 4 reports the resulting ablation. The skill-conditioned agent improves the overall construction success rate from 42.1% to 69.6%, with gains across all eight ecosystems. This supports the design choice of keeping the construction pipeline shared while injecting language-specific skills for repository exploration, dependency resolution, and test execution.

Table 4: Ablation of language skills in Stage-2 construction. Rates are instance-level construction success rates (%) measured on a fixed subset of candidate instances for each language.

Language	No-skill agent Rate	Skill agent Rate	Δ pp
C/C++	29.5	32.9	+3.4
Go	51.7	90.4	+38.6
Java	52.1	86.2	+34.1
JS/TS	27.8	85.0	+57.2
PHP	46.3	61.5	+15.2
Python	42.0	68.9	+27.0
Ruby	36.6	80.4	+43.8
Rust	49.0	77.4	+28.4
Overall	42.1	69.6	+27.5

F Language Skills Used in Construction

This appendix documents the Stage-2 construction skills used by SWE-CRAFTER. The default skill encodes cross-language construction rules, and each language skill supplies ecosystem-specific priors for repository evidence, runtime setup, target-test selection, common repair actions, and drop conditions. These skills are used as construction guidance rather than as ground truth; repository-local evidence and executable validation remain the final decision criteria.

Default SWE-Crafter Skill

```
Use this default skill together with the language-specific skill. It encodes cross-language construction rules that
      should hold for every Stage-2 task.
## Evidence Priority
1. Prefer repository-local evidence over generic language habits.
2. Inspect, in order: README or developer docs, CI workflows, package manifests, lock files, test configuration, scripts,
       and recent setup files.
3. If the issue or test patch names a specific test file, class, function, or fixture, treat that as the target-test
      scope.
4. Record the evidence used for runtime versions, dependency commands, and test selection in the construction notes or
      status payload.
## Dockerfile Rules
- Start from a deterministic Debian/Ubuntu-compatible base unless repository evidence clearly requires a specific Debian/
      Ubuntu language image.
- Clone the repository at `base_commit` into `/testbed`, remove remote origin, mark `/testbed` as a safe Git directory,
      and keep permissions writable.
- Install system packages before language dependencies. Common base packages include `git`, `curl`, `jq`, `patch`, `build
      -essential`, `pkg-config`, `ca-certificates`, and locale/timezone data.
- Do not copy repository contents from local disks or temporary images.
- Do not run tests in the Dockerfile. Build dependencies there; execute tests only through `eval.sh`.
## eval.sh Rules
- Check out target test files at the base commit, apply the complete `test_patch` through a heredoc, and run only target
      tests when possible.
- Activate the environment created in the Dockerfile before running tests.
- Capture the test exit code immediately with `rc=$?` and print `OMNIGRIL_EXIT_CODE=$rc`.
- Reset target test files after execution to avoid polluting later runs.
- Use fresh containers for construction-time before/after checks. Final dataset acceptance is still decided by the
      external strict F2P verifier.
## Repair and Drop Conditions
- If a failure points to missing native libraries, runtime mismatch, or an uninitialized service, inspect repo/CI
      evidence before changing commands.
- If target tests cannot be selected, prefer the smallest documented subset rather than the whole suite.
- Drop the candidate if the repository cannot build, target tests are not discoverable, tests are flaky under repeated
      fresh containers, the base run already passes, or the golden-patch run cannot pass within the repair budget.
```

Python Skill

```
## Evidence Priority
- Version evidence: `.python-version`, `runtime.txt`, `pyproject.toml`, `setup.cfg`, `setup.py`, `tox.ini`, `noxfile.py`,
       `uv.lock`, `poetry.lock`, `Pipfile.lock`, `requirements*.txt`, CI matrices, and README install docs.
- Test evidence: `pytest.ini`, `pyproject.toml` `[tool.pytest.ini_options]`, `tox.ini`, `noxfile.py`, `Makefile`, and
      scripts under `scripts/` or `ci/`.
- Prefer project docs and CI over generic `pytest` assumptions.
## Runtime and Dependency Setup
- Use the Python version pinned by repo evidence. If multiple versions appear, prefer the CI version used by tests that
      touch the target files.
- Choose the package manager from repo evidence: `uv` for `uv.lock` or explicit docs, Poetry for `poetry.lock`, Pipenv
      for `Pipfile.lock`, otherwise pip.
- Install native packages before Python dependencies when logs mention headers or shared libraries, e.g. `libpq-dev`, `
      libffi-dev`, `libssl-dev`, `libxml2-dev`, `libxslt1-dev`, `zlib1g-dev`, `libjpeg-dev`, `gdal-bin`, or `graphviz`.
- Use editable installs only when tests import the local package from the checkout. Common patterns are `pip install -e
      .`, `pip install -e .[test]`, `uv pip install -e ".[dev]"`, or `poetry install --with dev`.
- If dependency resolution is slow or impossible, follow lock files first; do not upgrade major dependencies unless the
      repository docs require it.
## Target-Test Commands
- Prefer the repository's documented command. Common forms:
  - `python -m pytest <target_file>`
  - `pytest <target_file>::<test_name>`
  - `tox -e <env> -- <target_file>`
  - `nox -s <session> -- <target_file>`
  - `python -m unittest <module_or_class>`
- For monorepos, run from the package directory that owns the target test and set `PYTHONPATH` only when local docs or
      imports require it.
- Avoid full-suite commands unless target tests cannot be selected.
## Common Failures and Repairs
- `ModuleNotFoundError`: check editable install, package extras, and current working directory before adding `PYTHONPATH
      `.
- `ImportError` for native modules: install system headers and rebuild wheels.
- `pytest` collection mismatch: inspect test config and run from the repo root or package root expected by CI.
- Database, Redis, browser, or network fixtures: prefer lightweight local services or skip only if repo docs mark them
      optional; otherwise drop.
- Flaky async/time tests: rerun in a fresh container before accepting.
## Drop Conditions
- Drop if the target tests require unavailable external credentials, proprietary services, GPU-only runtime, or non-
      deterministic network access.
- Drop if the base+test-patch run already passes or if the golden-patch run cannot pass in a fresh container after
      following repository evidence.
```

JavaScript/TypeScript Skill

```
## Evidence Priority
- Version evidence: `.nvmrc`, `.node-version`, `package.json` `engines`, `volta`, `corepack`, CI matrices, Dockerfiles,
      and README setup sections.
- Package manager evidence: `pnpm-lock.yaml`, `yarn.lock`, `package-lock.json`, `bun.lockb`, `npm-shrinkwrap.json`,
      workspace files, and CI install commands.
- Test evidence: `package.json` scripts, Jest/Vitest/Mocha/Ava configs, Playwright/Cypress configs, Turborepo/Nx/Lerna
      workspace metadata, and target test file paths.
## Runtime and Dependency Setup
- Use the pinned Node version. If no pin exists, choose the lowest modern LTS compatible with `engines` and CI.
- Enable Corepack when the repo uses pnpm or Yarn Berry. Do not mix package managers unless repo docs do so.
- Install dependencies with the lockfile-compatible command: `npm ci`, `pnpm install --frozen-lockfile`, `yarn install --
      immutable`, or `bun install --frozen-lockfile`.
- For native modules such as `canvas`, `sharp`, `sqlite3`, `node-sass`, `esbuild`, or `puppeteer`, install required
      system libraries before package install.
- In monorepos, identify the owning workspace and run installs/tests at the level expected by the package manager.
## Target-Test Commands
- Prefer package scripts over direct runner invocation:
  - `npm test -- <target_file>`
  - `pnpm test -- <target_file>`
  - `yarn test <target_file>`
  - `npm run test:unit -- <target_file>`
- For Jest/Vitest/Mocha:
  - `npx jest <target_file> --runInBand`
  - `npx vitest run <target_file>`
  - `npx mocha <target_file>`
- For workspaces, use filters when available: `pnpm --filter <pkg> test`, `yarn workspace <pkg> test`, `npx turbo run
      test --filter=<pkg>`, or `npx nx test <pkg>`.
- Avoid browser/e2e suites unless the target patch modifies e2e tests and repo docs provide a headless setup.
## Common Failures and Repairs
- `ERR_PNPM_LOCKFILE_CONFIG_MISMATCH`: use the repo's pnpm version or Corepack.
- TypeScript path failures: inspect `tsconfig*.json`, workspace root, and build steps such as `tsc -b` or generated code.
- Native build failures: add `python3`, `make`, `g++`, `pkg-config`, and library dev packages before install.
- Snapshot failures: ensure the test command does not update snapshots.
- Timeouts from watchers: use non-watch modes such as `vitest run` or `jest --runInBand`.
## Drop Conditions
- Drop when tests require cloud credentials, a browser stack that cannot run headlessly, or proprietary package
      registries.
- Drop when dependency resolution cannot be reproduced from public lockfiles and repository docs.
```

Java Skill

```
## Evidence Priority
- Version evidence: `.java-version`, `pom.xml`, `build.gradle*`, `gradle/wrapper/gradle-wrapper.properties`, Maven
      toolchains, CI matrices, Dockerfiles, and README setup instructions.
- Build evidence: Maven/Gradle wrapper scripts, module layout, parent POMs, profiles, generated-source plugins,
      annotation processors, and integration test conventions.
- Test evidence: Surefire/Failsafe config, Gradle test tasks, package/class names from the test patch, and module
      ownership.
## Runtime and Dependency Setup
- Prefer repository wrappers: `./mvnw` or `./gradlew`. Use system Maven/Gradle only when wrappers are absent.
- Match the JDK version used by CI. If Maven/Gradle reports source/target compatibility, choose a JDK that satisfies it.
- Cache dependencies inside the image only after repository checkout so wrapper scripts and build files are available.
- Install native tools only when required by plugins or tests, e.g. `protobuf`, `graphviz`, `nodejs`, `python3`, `
      libsqlite3-dev`, or `docker-cli`.
- For Android projects, prefer command-line Gradle tasks and avoid emulator requirements; drop if an emulator or
      proprietary SDK is required.
## Target-Test Commands
- Maven:
  - `./mvnw -pl <module> -am test -Dtest=<ClassName>#<method>`
  - `./mvnw -pl <module> -am -DskipITs test`
  - `./mvnw -pl <module> -am -DfailIfNoTests=false test`
- Gradle:
  - `./gradlew :<module>:test --tests '<package.Class.method>'`
  - `./gradlew test --tests '<ClassName>'`
  - Add `--no-daemon --stacktrace` for deterministic logs.
- For integration tests, use the repository profile/task only if target files are integration tests and services can be
      started locally.
## Common Failures and Repairs
- `invalid target release`: switch JDK to match source/target or toolchain.
- Missing generated classes: run required generate-sources task or include module dependencies with `-am`.
- `No tests found`: verify class name, module path, JUnit version, and test selector syntax.
- Gradle daemon or memory failures: use `--no-daemon` and set conservative `GRADLE_OPTS` or `MAVEN_OPTS`.
- Repository uses Lombok/annotation processors: ensure plugin repositories and processor dependencies resolve before
      tests.
## Drop Conditions
- Drop if target tests require external enterprise services, unavailable proprietary artifacts, Android emulators, or non
      -public Maven repositories.
- Drop if the smallest target-test command cannot be isolated and the full build exceeds the repair/runtime budget.
```

Go Skill

```
## Evidence Priority
- Version evidence: `go.mod` `go` and `toolchain` directives, `.go-version`, CI matrices, Dockerfiles, and README setup
      docs.
- Build evidence: `Makefile`, `magefile.go`, `Taskfile.yml`, generated-code scripts, `tools.go`, and module/workspace
      layout.
- Test evidence: target package path, test file names, build tags, integration flags, and fixtures referenced by the test
       patch.
## Runtime and Dependency Setup
- Use the Go version indicated by `go.mod` or CI. Respect `GOTOOLCHAIN` when the repository expects a newer toolchain.
- Run `go mod download` or the documented setup command after installing native dependencies.
- Install CGO libraries before tests when packages reference SQLite, libgit2, RocksDB, PCRE, ImageMagick, or other native
       bindings.
- If generated files are required, run the documented `make generate`, `go generate ./...`, `buf generate`, `protoc`, or
      mock-generation command.
- Set `CGO_ENABLED` only according to repository evidence; do not disable CGO to bypass missing native dependencies if
      tests need it.
## Target-Test Commands
- Prefer package-level tests for the package containing target files:
  - `go test ./pkg/foo`
  - `go test ./pkg/foo -run 'TestName'`
  - `go test ./pkg/foo -run 'TestName/Subcase'`
- Include build tags or flags from CI/docs: `go test -tags '<tags>' ./pkg/foo -run 'TestName'`.
- For workspaces, run from the module root that owns the package and set `GOWORK` only if the repo uses `go.work`.
- Avoid `go test ./...` unless target package selection is impossible.
## Common Failures and Repairs
- `missing go.sum entry`: run `go mod download` or the repo's setup command.
- `build constraints exclude all Go files`: inspect build tags and OS/arch assumptions.
- Generated mock/protobuf errors: run the generator documented by the repo.
- Integration tests fail on services: inspect docker-compose docs; use local service only if it is lightweight and
      documented.
- Race/time-sensitive tests: avoid `-race` unless CI target requires it.
## Drop Conditions
- Drop if tests require cloud credentials, privileged kernel features, large external services, or private modules.
- Drop if generated code depends on unavailable proprietary tooling.
```

C/C++ Skill

```
## Evidence Priority
- Build evidence: README build docs, CI workflows, `CMakeLists.txt`, `Makefile`, `configure.ac`, `meson.build`, `BUILD`/
      Bazel files, `conanfile.*`, `vcpkg.json`, submodule config, and package manager manifests.
- Version evidence: compiler standards in CMake/Meson/Bazel, CI compiler matrix, distro packages, and toolchain files.
- Test evidence: CTest labels, GoogleTest/Catch2/doctest test names, target test files, and custom scripts under `test/`,
       `tests/`, or `ci/`.
## Runtime and Dependency Setup
- Install compilers and build tools first: `build-essential`, `cmake`, `ninja`, `make`, `pkg-config`, `autoconf`, `
      automake`, `libtool`, and `python3` when generators require it.
- Respect the repository build system. Do not convert between CMake, Make, Meson, Autotools, or Bazel unless docs
      explicitly support it.
- Initialize submodules only when repository evidence requires them.
- Install native libraries from CI/docs before configure/build, e.g. OpenSSL, zlib, Boost, protobuf, sqlite, curl,
      libxml2, libpng/jpeg, LLVM/Clang, or Qt/GTK development packages.
- Keep build directories out of source when supported, e.g. `build/`.
## Target-Test Commands
- CMake/CTest:
  - `cmake -S . -B build -G Ninja <flags>`
  - `cmake --build build --target <target>`
  - `ctest --test-dir build -R '<regex>' --output-on-failure`
- Make/Autotools:
  - `./configure <flags> && make -j$(nproc)`
  - `make <test_target>` or `make check TESTS='<target>'`
- Meson:
  - `meson setup build <flags> && meson test -C build <test_name>`
- Bazel:
  - `bazel test //<package>:<target>` with repo-documented flags.
- Prefer test-target or regex selection over full project test suites.
## Common Failures and Repairs
- Missing headers/libraries: inspect CI apt packages and `pkg-config` names.
- C++ standard errors: match compiler and `-std=` flags from CI/toolchain.
- Submodule errors: run documented `git submodule update --init --recursive`.
- Generated code missing: install protobuf/flex/bison/python generators and run the documented generation step.
- Linker failures: ensure dev packages and runtime libraries are installed in the same image stage.
## Drop Conditions
- Drop if tests require GPUs, kernel modules, privileged hardware, proprietary SDKs, or very large third-party downloads.
- Drop if only full-system integration tests exist and cannot run in a standard container within the repair/runtime
      budget.
```

Rust Skill

```
## Evidence Priority
- Version evidence: `rust-toolchain`, `rust-toolchain.toml`, `Cargo.toml` `rust-version`, CI matrices, Dockerfiles, and
      README setup docs.
- Build evidence: workspace layout, feature flags, `build.rs`, generated bindings, `justfile`, `Makefile`, and scripts
      under `xtask/` or `tools/`.
- Test evidence: target crate, integration test file, unit test module, required features, and CI test flags.
## Runtime and Dependency Setup
- Install the pinned Rust toolchain with rustup when repo evidence requires a specific stable/nightly version.
- Run `cargo fetch` after installing native dependencies.
- Install system libraries for crates that bind to OpenSSL, SQLite, zlib, libclang, protobuf, libgit2, rocksdb, or image/
      font stacks.
- If `build.rs` requires code generation, install `clang`, `llvm`, `protoc`, `cmake`, or other documented tools before `
      cargo test`.
- Do not remove default features unless the target test explicitly uses a feature-reduced configuration.
## Target-Test Commands
- Prefer the smallest owning crate/package:
  - `cargo test -p <package> <test_name>`
  - `cargo test -p <package> --test <integration_test>`
  - `cargo test -p <package> --features '<features>' <test_name>`
  - `cargo test --workspace --exclude <large_pkg>` only when needed.
- For doctests or examples, use the command documented by CI.
- Use `-- --nocapture` only when logs are needed for debugging.
## Common Failures and Repairs
- Toolchain mismatch: install the version pinned by `rust-toolchain*` or `rust-version`.
- `failed to run custom build command`: inspect `build.rs` and install native libraries or generators.
- Feature-gated items missing: add features from CI/docs or target crate metadata.
- Linker errors: install `pkg-config` and relevant `*-dev` packages, then rerun.
- Workspace package not found: run from workspace root and use exact package name from `Cargo.toml`.
## Drop Conditions
- Drop if target tests require unavailable hardware, privileged kernel access, networked services without local setup, or
       private crates.
- Drop if the crate cannot compile with any toolchain supported by repository evidence within the repair budget.
```

Ruby Skill

```
## Evidence Priority
- Version evidence: `.ruby-version`, `.tool-versions`, `Gemfile`, `Gemfile.lock`, CI matrices, Dockerfiles, and README
      setup docs.
- Dependency evidence: Bundler version, Rails/Sinatra/Rake configuration, native gem dependencies, and database/service
      setup docs.
- Test evidence: `spec/`, `test/`, `.rspec`, `Rakefile`, Minitest/RSpec config, and target file/example names from the
      test patch.
## Runtime and Dependency Setup
- Use the Ruby version pinned by repo evidence. Install a compatible Bundler version when `Gemfile.lock` has `BUNDLED
      WITH`.
- Run `bundle install` with path/cache options suitable for containers. Avoid `bundle update`.
- Install native libraries before bundle install for gems such as `pg`, `mysql2`, `sqlite3`, `nokogiri`, `ffi`, `grpc`, `
      rmagick`, or `charlock_holmes`.
- For Rails apps, prepare test environment from repo docs: copy example env files, use SQLite when documented, and run
      migrations only for test DB.
- Do not start long-lived app servers unless tests explicitly require them.
## Target-Test Commands
- RSpec:
  - `bundle exec rspec <target_file>`
  - `bundle exec rspec <target_file>:<line>`
  - `bundle exec rspec --example '<example text>'`
- Minitest:
  - `bundle exec ruby -Itest <target_file>`
  - `bundle exec rake test TEST=<target_file>`
- Rails:
  - `bundle exec rails test <target_file>`
  - `RAILS_ENV=test bundle exec rspec <target_file>`
- Prefer file/line/example selectors over full `bundle exec rake test`.
## Common Failures and Repairs
- Bundler version mismatch: install the exact Bundler version from lockfile.
- Native gem build errors: install headers and libraries before `bundle install`.
- Database errors: inspect `config/database.yml`, examples, and CI; prefer documented SQLite/test DB setup.
- Zeitwerk/load errors: run from repo root and ensure bundle install includes test/development groups when required.
- Timezone/locale failures: set `TZ=Etc/UTC` and install locales.
## Drop Conditions
- Drop if target tests require proprietary services, unavailable private gems, browser/e2e infrastructure without
      headless setup, or non-public credentials.
- Drop if framework boot cannot be reproduced from public docs and configs.
```

PHP Skill

```
## Evidence Priority
- Version evidence: `composer.json` `require.php`, `composer.lock`, `.php-version`, CI matrices, Dockerfiles, and README
      setup docs.
- Dependency evidence: Composer scripts, PHPUnit/Pest configs, Symfony/Laravel console commands, extension requirements,
      and service fixtures.
- Test evidence: `phpunit.xml*`, `pest.php`, target test file/class/method, and framework-specific test docs.
## Runtime and Dependency Setup
- Use the PHP version compatible with Composer constraints and CI.
- Install required PHP extensions before `composer install`: common extensions include `mbstring`, `xml`, `curl`, `zip`,
      `intl`, `gd`, `pdo_mysql`, `pdo_pgsql`, `sqlite3`, `bcmath`, `redis`, and `imagick`.
- Prefer `composer install --no-interaction --prefer-dist` with lockfile preservation. Do not run `composer update`.
- If the project is a framework app, prepare `.env` only from repo examples and documented test configuration; never add
      private credentials.
- For Laravel/Symfony, run documented cache/config/bootstrap commands only when tests require them.
## Target-Test Commands
- PHPUnit:
  - `vendor/bin/phpunit <target_file>`
  - `vendor/bin/phpunit --filter '<testMethod>' <target_file>`
  - `composer test -- <target_file>` when scripts wrap PHPUnit.
- Pest:
  - `vendor/bin/pest <target_file>`
  - `vendor/bin/pest --filter '<testName>'`
- Framework commands:
  - `php artisan test <target_file>` for Laravel when docs use it.
  - Symfony projects may use `vendor/bin/phpunit` with `APP_ENV=test`.
- Avoid full test suites unless target selection is unsupported.
## Common Failures and Repairs
- Composer platform errors: install matching PHP extensions or set the PHP version to the repo-supported one; do not
      ignore platform requirements unless docs explicitly do so.
- Autoload failures: run `composer dump-autoload` after install.
- Database failures: prefer SQLite in-memory or documented test database setup.
- Missing app key/cache: run documented test bootstrap such as `php artisan key:generate --env=testing` only when
      required.
- Timezone/locale failures: set `TZ=Etc/UTC` and install locales.
## Drop Conditions
- Drop if tests require private Composer repositories, external paid services, unavailable PHP extensions, or a full web
      server stack that cannot be reproduced locally.
- Drop if base+test-patch already passes or golden-patch cannot pass after following Composer and framework evidence.
```

G Limitations and Responsible Release

The construction pipeline relies on public GitHub issue–pull-request pairs, repository-level tests, and automatically synthesized execution environments. It can therefore underrepresent projects whose tests are flaky, unavailable, or difficult to execute in containers, and it may inherit coverage biases from public open-source repositories.

The released data are intended to support research on executable SWE tasks. Public upstream repository identifiers and pinned commits provide attribution, license traceability, and a basis for reconstructing repository states. Dockerfiles and test commands document the execution environment for each task. Researchers can use these records to inspect supervision and study task-level verification across languages.

Responsible redistribution requires attention to personal contact information, credentials, private machine paths, and infrastructure-specific metadata that may occur in repository-derived text. The release policy calls for privacy review of these fields while preserving public provenance needed for attribution and verification. Users should obtain upstream source code from the referenced repositories and respect the applicable repository, dependency, and model terms. The dataset documentation describes the intended use and these privacy and licensing considerations.

H Training and Compute Details

We train SWE-CRAFTER models with supervised fine-tuning on test-passing teacher-model trajectories. Following common SWE-agent fine-tuning practice [48], we use MS-Swift [28] for full-parameter supervised fine-tuning with the AdamW optimizer, a learning rate of 5×10^{-5}, a maximum of 3 epochs, and a maximum context length of 128K tokens. The cross-language transfer and data scaling studies use the same training recipe, changing only the language or number of sampled training trajectories. Evaluation uses OpenHands with the same single-attempt benchmark harness described in Section 3.

The main supervised fine-tuning runs are conducted on an internal GPU cluster. With a 512-GPU training allocation, each main run takes approximately 8 hours. The full data-construction pipeline additionally requires containerized repository builds, test execution, and teacher-model rollouts; these costs depend on repository build time and test-suite runtime.

I Existing Assets and Licenses

SWE-CRAFTER builds instances from public GitHub repositories and evaluates on public SWE benchmarks. We cite the benchmarks, model families, training tools, and agent frameworks used in the paper, including SWE-bench Multilingual, SWE-bench Verified, MiniMax M2.5, Qwen, IQuest-Coder, MS-Swift, and OpenHands. The constructed records preserve upstream repository identifiers and commits to retain attribution and license traceability. We do not redistribute complete upstream repository snapshots; users should reconstruct source code from the original repositories and comply with the corresponding repository licenses and terms of use. We also apply license-based filtering during repository selection. Repositories without an explicit license, or with license terms incompatible with our intended research release and redistribution setting, are removed. The retained repositories are restricted to MIT, Apache-2.0, BSD-3-Clause, and BSD-2-Clause licenses.

The dataset documentation describes the task format, provenance, intended research use, and licensing considerations. Repository-derived issue text, patches, and tests remain subject to the applicable upstream terms; the referenced repository and commit identify the corresponding source and license. Privacy review should cover personal contact information, access tokens, API keys, password-like or credential-like strings, private machine paths, and internal storage or execution paths. Users who rebuild environments from the task Dockerfiles and test commands remain responsible for complying with the licenses and terms of the upstream repositories and dependencies they install.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes].

   Justification: The abstract and introduction state the framework, the constructed data scale, the model-training contribution, and the analysis scope. The claims are tied to executable multilingual SWE instance construction and evaluation on SWE-bench Multilingual and SWE-bench Verified.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes].

   Justification: The appendix states that the construction pipeline relies on public GitHub issue–pull-request pairs, repository-level tests, and automatically synthesized execution environments. It also notes that this can underrepresent projects with flaky, unavailable, or container-difficult tests and inherit coverage biases from public open-source repositories.

   Guidelines:

- The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
- The authors are encouraged to create a separate "Limitations" section in their paper.
- The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
- The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
- The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
- The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
- If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
- While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A].

   Justification: The paper is empirical and does not present theoretical theorems or proofs.

   Guidelines:

   - The answer [N/A] means that the paper does not include theoretical results.
   - All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
   - All assumptions should be clearly stated or referenced in the statement of any theorems.
   - The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
   - Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
   - Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes].

   Justification: Appendix C describes a data-based verification route using the released task records, pinned repository states, patches, Dockerfiles, and test commands. Section 2 describes construction and filtering, and Section 3 and Appendix H report the teacher model, trajectory selection, student training recipe, and benchmark protocol.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes].

   Justification: Appendix B provides the released dataset, and Appendix C describes task reconstruction and Fail-to-Pass verification using the records. Model training and benchmark evaluation use the public MS-Swift and OpenHands frameworks with the settings reported in Section 3 and Appendix H.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.

- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes].

   Justification: The main paper specifies the benchmarks, teacher model, base models, training data source, strict Fail-to-Pass filtering before distillation, MS-Swift training scaffold, OpenHands evaluation scaffold, and single-attempt metric. The appendix further specifies the supervised fine-tuning optimizer, learning rate, epoch count, context length, and scaling-study controls.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No].

   Justification: The main results are reported as single-attempt resolved rates on repository-level benchmarks. We do not report repeated-run confidence intervals or error bars because evaluation requires costly model inference and containerized repository execution.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).
   - It should be clear whether the error bar is the standard deviation or the standard error of the mean.
   - It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
   - For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
   - If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes].

Justification: The appendix reports that each main supervised fine-tuning run takes approximately 8 hours with a 512-GPU training allocation, and it describes the additional containerized repository execution and teacher-rollout costs involved in data construction.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
- The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
- The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

Answer: [Yes].

Justification: The work uses public software repositories and repository-level evaluation without human-subject experiments. The appendix describes the released task data, the privacy review policy for repository-derived records, and upstream-license considerations.

Guidelines:

- The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
- If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
- The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes].

Justification: The appendix explains that the data is intended to support research on executable SWE training instances and also discusses privacy and licensing considerations for repository-derived records, including privacy review, preservation of public provenance, and reconstruction of source code from upstream repositories.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.

- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [Yes].

    Justification: Appendix G describes the privacy review policy for repository-derived task records, including attention to personal contact information, credentials, private paths, and infrastructure metadata. It also explains preservation of public provenance for attribution and verification and respect for upstream repository, dependency, and model terms.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes].

    Justification: The paper cites the major datasets, benchmarks, models, and agent frameworks it uses, and the constructed records preserve upstream repository identifiers and commits for attribution and license traceability. The appendix states that repository selection applies license-based filtering and retains only MIT, Apache-2.0, BSD-3-Clause, and BSD-2-Clause licensed repositories. It also clarifies that Dockerfiles and evaluation scripts do not redistribute third-party dependency source code or full upstream repository snapshots.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
    - For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
    - If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

Answer: [Yes].

Justification: The dataset linked in Appendix B is documented through the task format and construction procedure in Section 2, the verification protocol in Appendix C, and the intended use, limitations, provenance, and licensing considerations in Appendices G and I. The accompanying dataset card summarizes these details alongside the data.

Guidelines:

- The answer [N/A] means that the paper does not release new assets.
- Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
- The paper should discuss whether and how consent was obtained from people whose asset is used.
- At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

Answer: [N/A].

Justification: The paper does not involve crowdsourcing experiments or research with human subjects.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
- According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A].

Justification: The paper does not involve crowdsourcing experiments or research with human subjects, so IRB approval is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes].

Justification: The method section describes the use of LLM-based agents and a teacher model for instance construction, executable validation, trajectory distillation, and supervised fine-tuning.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.