SWE-AT-SCALE: SCALABLE ISSUE SYNTHESIS FROM HISTORICAL BUG FIXES

Anonymous authors
Paper under double-blind review

ABSTRACT

As coding agents become increasingly capable, robust evaluation requires a broader and more diverse supply of verifiable software engineering (SWE) tasks. However, real-world tasks are constrained by the natural frequency of issue reporting and the substantial cost of manual curation. Furthermore, existing synthetic approaches rely primarily on the parametric knowledge of large models coupled with heuristic prompting, often failing to capture the authentic, complex mechanics of software failures. To exponentially scale SWE task production, we introduce SWE-at-Scale, a framework that automatically synthesizes realistic issues by reverse-engineering real-world software fixes. Our key insight is that historical fixes encapsulate deep mechanistic knowledge of bug formation: inverting a fix transforms repair supervision into bug-induction supervision grounded in actual development scenarios. SWE-at-Scale transfers these learned bug patterns to novel code contexts, repeatedly sampling diverse bug typologies to construct executable SWE tasks. Extensive experiments show that SWE-at-Scale generates more diverse and realistic SWE tasks than current synthesis baselines while posing greater challenges to state-of-the-art coding agents. Notably, SWE-at-Scale can generate distinct bug types from a single code context, with bug diversity scaling robustly even up to 200 generations per file. Overall, SWE-at-Scale turns finite real-world fixes into reusable supervision for scalable synthesis of realistic, diverse, and challenging SWE tasks.

1 INTRODUCTION

Repository-level software engineering (SWE) tasks have become a central testbed for evaluating coding agents (Chen et al., 2025b; Lin et al., 2024; Ma et al., 2026; Chen et al., 2026b;c; Lin et al., 2026; Gu et al., 2025; Shi et al., 2026a; Shao et al., 2025). A useful SWE task involves more than a code change (Yang et al., 2025; Sonwane et al., 2025): it should preserve a realistic repository context, introduce an observable behavioral bug, provide a meaningful issue description, and remain executable within the repository's testing infrastructure.

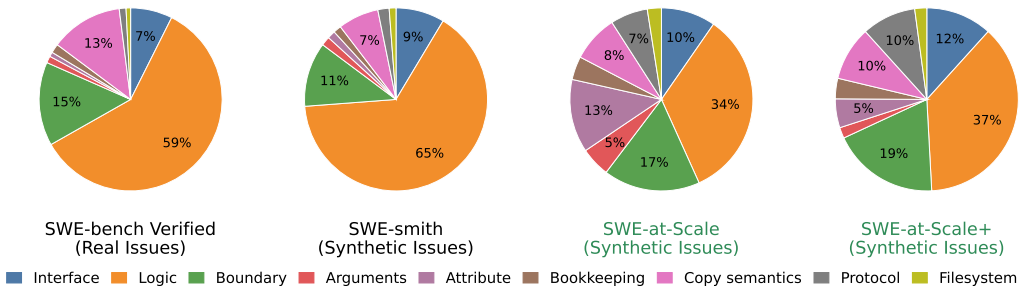


Figure 1: Bug type distributions for real-world SWE-bench Verified tasks, SWE-smith synthetic tasks, and SWE-at-Scale synthetic SWE tasks.

Existing real-world SWE benchmarks (Jimenez et al., 2024) are inherently difficult to scale, as new tasks can only emerge at the pace of real software development (Zhang et al., 2025). Collecting

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

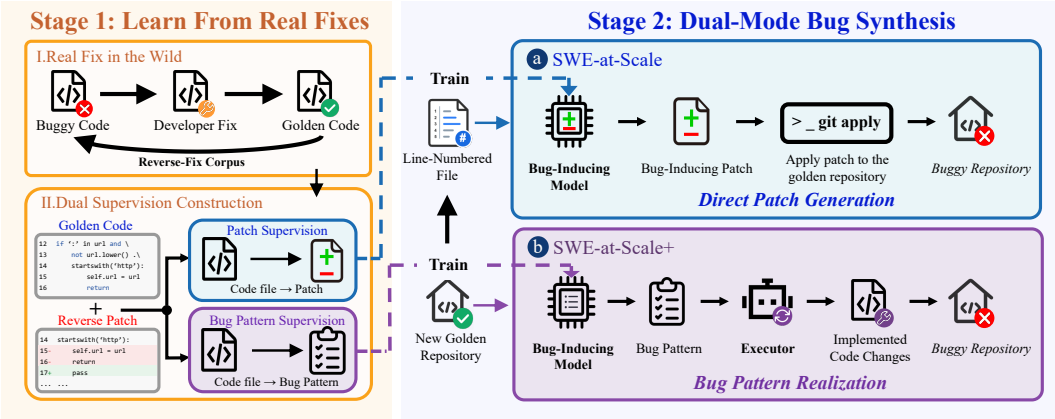


Figure 2: Overview of SWE-at-Scale. Stage 1 reverses real software fixes to construct bug-inducing patch and bug-pattern supervision, which is used to train the synthesis models. Stage 2 applies the learned transformations to new golden repositories through two complementary synthesis models.

these tasks is further constrained by repository availability and the cost of constructing executable environments (Jain et al., 2025). Consequently, scaling SWE task data solely through naturally occurring issues requires continuously discovering and validating additional real-world development events.

Recent work has explored large language models (LLMs) as a means of scaling SWE task construction. SWE-smith (Yang et al., 2025) leverages LLMs to inject bugs by prompting them to modify selected code entities, enabling the synthesis of large numbers of synthetic task instances. SWE-Synth (Pham et al., 2025) employs LLM-based component re-implementation followed by test-based filtering, while BugPilot (Sonwane et al., 2025) explores feature addition and behavioral regression as a source of synthetic bugs. These approaches demonstrate that LLMs can generate bugs beyond simple hand-designed mutation rules. However, without dedicated training for bug injection, these approaches largely depend on the bug-generation knowledge already encoded in general-purpose LLMs and elicited through prompting, making repeated generation more likely to revisit familiar bug patterns rather than continually expanding the diversity of synthesized tasks. This motivates a different scaling strategy: rather than repeatedly prompting a general-purpose LLM, we reverse real-world bug fixes to construct correct-to-buggy supervision and train a reusable bug-injection model that can generate diverse bugs in new code contexts.

We introduce SWE-at-Scale, a framework that turns real software fixes into a scalable source of repository-level SWE tasks. Rather than treating each historical fix as a single, one-off task instance, SWE-at-Scale reverses bug-fixing transformations and uses them as supervision for learning how bugs can be induced. Aggregating such reversed transformations across diverse repositories allows the model to learn a reusable distribution of real-world bug patterns, instead of relying solely on the bug-generation knowledge already encoded in general LLMs. Through this reverse-learning formulation, SWE-at-Scale can repeatedly sample these real-world bug patterns and instantiate them in new code contexts, allowing a finite collection of real fixes to support the synthesis of a much larger and more diverse set of executable SWE tasks.

We train SWE-at-Scale and SWE-at-Scale+ on 13,407 real-fix transformations collected from 688 repositories, using Qwen3.5-9B as the backbone model. SWE-at-Scale directly learns to generate bug-inducing patches, while SWE-at-Scale+ explicitly learns natural-language bug patterns that describe how the target code should be modified to induce a bug. Experiments show that SWE-at-Scale synthesizes SWE tasks that are realistic, challenging, diverse, and scalable under repeated generation. In pairwise comparisons with real-world SWE-bench Verified issues, human evaluators select SWE-at-Scale- and SWE-at-Scale+-generated bugs as more likely to occur in real software development in 55.9% and 61.2% of cases, respectively. The synthesized tasks also remain challenging for current coding agents: across four resolution models, SWE-at-Scale achieves Pass@1 between 31.68% and 48.48%, while SWE-at-Scale+ achieves between 16.81% and 37.02%, with SWE-at-Scale+ approaching the difficulty of human-expert-designed tasks such as DeepSWE (Huang et al., 2026). As shown in Figure 1, SWE-at-Scale covers a more balanced range of bug types than existing

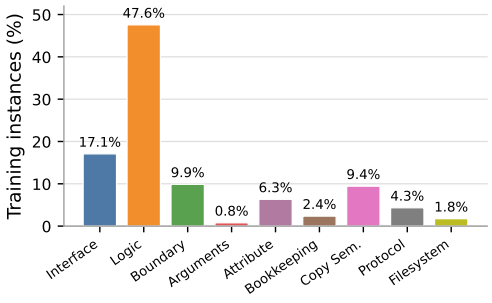


Figure 3: Bug type distribution in the training dataset.

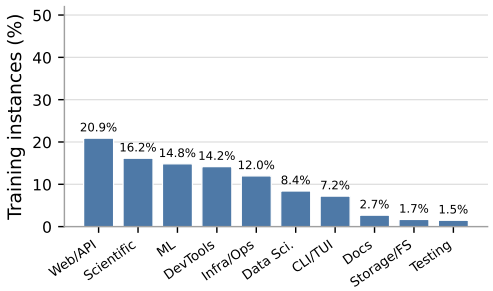


Figure 4: Distribution of training instances across repository domain categories.

synthesis methods, suggesting that learning from real fixes captures a richer bug distribution than relying on general LLMs' knowledge alone. This diversity also persists under repeated sampling: even after 200 generations per file, 88.27% of usable patches remain unique, indicating that the generated bugs do not quickly collapse to repeated outcomes.

Our contributions are threefold: 1) We introduce SWE-at-Scale, a scalable paradigm for SWE task synthesis. SWE-at-Scale reverses real bug fixes as supervision for learning how bugs are introduced, allowing bug patterns observed in real development to be instantiated in new code contexts; 2) We construct a large-scale training corpus of 13,407 real-fix transformations from 688 repositories and develop two synthesis variants: SWE-at-Scale directly learns bug-inducing patches, while SWE-at-Scale+ externalizes bug-inducing logic as natural-language patterns for more explicit realization; and 3) Through controlled comparisons with real-world and synthetic SWE tasks, we show that SWE-at-Scale produces realistic, diverse, and challenging executable tasks, and that its bug diversity continues to increase as the inference-time sampling budget grows.

2 SWE-at-Scale: Software Task Synthesis from Real Fixes

The core principle of SWE-at-Scale is to learn how bugs are introduced by reversing real software fixes. The learned bug patterns can be adapted to new code context, enabling the synthesis of large-scale SWE tasks. The pipeline involves two stages, as illustrated in Figure 2. Given a corpus of real bug-fix commits, SWE-at-Scale constructs reversed (code, buggy code) pairs (Stage I). A bug inducing model is trained using the collected pairs (Stage II). The trained model automatically generates buggy code given arbitrary source code. SWE-at-Scale learns these transformations and uses them to synthesize new SWE task instances.

2.1 Training Data Collection

To train the bug-inducing model, we construct a corpus of historical bug fixes and reverse them into (golden code, buggy code) pairs. We collect bug-fix instances from SWE-Fixer (Xie et al., 2025), SWE-rebench (Badertdinov et al., 2025), SWE-bench-Live (Zhang et al., 2025), and SWE-Gym (Pan et al., 2025b). The four sources contain more than 134K issue–fix instances in total. For each instance, we identify the source files modified by the real fix and recover their versions before and after the fix. For each real-fix instance, we first apply the original golden patch to the buggy repository to reconstruct the corresponding fixed source files. We then compute a new unified diff from each fixed file back to its original buggy version. This reverse-direction diff serves as the bug-inducing patch used for training SWE-at-Scale. To avoid repository-level overlap with our evaluation data, we exclude all instances from repositories represented in SWE-bench Verified.

We apply a series of quality filters to the collected records. We remove examples with missing or unparseable patches, non-code files, or changes dominated by imports, comments, whitespace, or formatting. We further discard structurally incomplete patches that do not represent a complete and internally consistent code transformation. For each retained record, we reconstruct the buggy and fixed file states, generate unified diffs in both directions, and verify that the forward and reverse patches apply successfully and reproduce the corresponding original states through round-trip ap-

plication. Following BugPilot (Sonwane et al., 2025), we annotate the remaining examples with bug types and remove examples whose serialized input–output sequences exceed the 64K-token limit.

From the remaining candidates, we construct the final training dataset through distribution-aware sampling, aiming to balance bug types, repositories, and context lengths as much as permitted by the available real-fix data. We further manually inspect the selected patches to ensure their quality. After filtering, reconstruction, distribution-aware sampling, and manual inspection, the final training dataset contains 13,407 real-fix transformations from 688 repositories, which provide the training supervision for both SWE-at-Scale synthesis models.

Figures 3 and 4 summarize the resulting data distribution. Our balancing strategy substantially broadens coverage across bug types, repositories, and software domains while preserving the naturally occurring characteristics of real software fixes. Logic and Interface transformations remain the most frequent categories, reflecting their prevalence in the real-fix data rather than an intentional sampling bias.

2.2 Training Bug-Inducing Model

SWE-at-Scale uses reversed real-fix transformations as supervision for training the bug inducing model. For each real-fix transformation, the fixed file serves as the golden input, while the reversed fix serves as the bug-inducing target. We use Qwen3.5-9B (Qwen Team, 2026) as the backbone model and fine-tune it to generate a unified diff from the complete golden source file annotated with line numbers. The generated diff transforms the correct code into its corresponding buggy state. Providing the complete file allows the model to infer the relevant edit locations from the surrounding code context rather than receiving predefined locations, thereby learning jointly where to modify the code and how to realize the corresponding bug-inducing changes. Unlike synthesis with general LLMs without dedicated bug-inducing training, this formulation explicitly trains the model on correct-to-buggy transformations observed in real software development.

At synthesis time, SWE-at-Scale can operate on any source code file in a correct repository state and generate a candidate bug-inducing patch, which is applied to the corresponding repository using `git apply`. Successfully applied patches produce candidate buggy repositories that are subsequently passed to execution-based validation. Sampling temperature controls how broadly SWE-at-Scale explores the learned bug-inducing distribution, enabling repeated sampling from the same code context for scalable task synthesis.

Direct patch generation provides a simple and efficient way to learn bug-inducing patches from reversed real-fix transformations, but it requires the model to jointly determine what bug to introduce and how to realize it directly in code. The underlying bug pattern therefore remains implicit in the generated patch. To study whether making this pattern explicit can provide stronger guidance for bug synthesis, we further introduce an enhanced variant, SWE-at-Scale+, which represents the intended bug transformation in natural language before realizing it in code. To construct supervision for SWE-at-Scale+, we use DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026) as an expert teacher. For each verified inverse transformation, the teacher is prompted to infer an executor-ready natural-language bug pattern that specifies the task category, target location, current behavior, specific code modification, implementation constraints, and expected observable effect. Using the same golden file as input, SWE-at-Scale+ is trained to predict this pattern instead of the unified-diff target used by SWE-at-Scale. At synthesis time, the generated bug pattern and golden code are provided to an executor that realizes the specified modifications. To minimize the influence of the executor's prior knowledge of bug generation, we present the pattern as ordinary code-modification instructions without revealing that the requested changes are intended to introduce a bug. The resulting code forms the candidate buggy repository for subsequent execution-based validation.

2.3 Task Assembly and Validation

Finally, SWE-at-Scale assembles SWE tasks using the generated code changes. Following SWE-smith (Yang et al., 2025) and BugPilot (Sonwane et al., 2025), we construct the final task through execution-based validation and issue-description generation. We first apply the generated code changes to the corresponding golden repository and execute its test suite. We retain only candidates that cause at least one previously passing test to fail, referred to as a Pass-to-Fail (P2F) test.

For each validated buggy repository, we generate a user-facing issue description without exposing the synthesis process. We provide an LLM with the generated code diff, the source code of a P2F test, its execution output and two demonstrations (Yang et al., 2025), and prompt it to produce a GitHub-style issue describing the observed and expected behavior. The resulting issue description is paired with the validated buggy repository to form the final SWE task instance.

3 EXPERIMENTS

3.1 SETUP

Models. For baseline task synthesis methods, we use DeepSeek-V4-Flash-0731 (DeepSeek-AI, 2026), GPT-5.4-mini, and Qwen3.5-9B (Qwen Team, 2026) as the underlying LLMs. For SWE-at-Scale+, we use DeepSeek-V4-Flash-0731 as the expert model for generating natural-language bug patterns, and Qwen3.5-9B as the base model for training both SWE-at-Scale and SWE-at-Scale+. The prompts are provided in our anonymous repository[1]. For downstream issue resolution, we use DeepSeek-V4-Flash-0731 (DS-V4-Flash), GPT-5.4-mini, GLM-5.3-Flash (GLM, 2026), and Qwen3.8-Flash-Next (Qwen3.8-Flash) (Qiu et al., 2026) as the coding agents to evaluate the difficulty of the synthesized task instances.

Baselines. We compare SWE-at-Scale against three representative synthesis methods: SWE-smith (Yang et al., 2025) with its LLM-Modify and LLM-Rewrite strategies, and BugPilot (Sonwane et al., 2025) with its FeatAdd strategy. We also include SWE-bench Verified (Jimenez et al., 2024) and DeepSWE (Huang et al., 2026) as a reference set of real-world SWE issues.

Implementation Details. We fine-tune SWE-at-Scale and SWE-at-Scale+ on the 13,407 real-fix transformations for one epoch using supervised fine-tuning (SFT) with LoRA (Hu et al., 2022) and a 64K-token context window. All training is performed on a single NVIDIA RTX PRO 6000 GPU; the complete LoRA and optimization configuration is provided in Appendix Table 2.

For task synthesis, each method receives up to five attempts per target file: the first uses temperature 0 and the remaining four use temperature 1.0. Generation terminates once a candidate causes at least one previously passing regression test to fail; otherwise, the instance is discarded after five attempts. For controlled comparison, all methods synthesize bugs from the same target file and golden repository state associated with each SWE-bench Verified instance. We use the default FeatAdd configuration from BugPilot. For SWE-smith, both LLM-Modify and LLM-Rewrite receive the complete target file and are constrained to the smallest enclosing functions or classes containing the golden-patch hunks. LLM-Rewrite masks and reimplements these units, whereas LLM-Modify keeps them visible and may introduce any number of modifications within them.

We evaluate synthesized tasks using mini-SWE-agent (Yang et al., 2024) with temperature 0 and a maximum of 250 actions. To avoid model-specific generation–resolution bias, we use model-disjoint evaluation: a model never resolves tasks whose problem statements were generated by itself. Each experimental configuration is run independently three times, and we report the mean across runs.

3.2 EVALUATION METRICS

We evaluate synthesized task instances in terms of realism, usability, and difficulty. We measure realism using the **Human Preference Rate (HPR)**, following the human evaluation protocol of CodeOCR (Shi et al., 2026b). Three software engineers with at least three years of experience compare each synthesized issue–patch pair with a real-world counterpart within the same context and judge which one they would be more likely to encounter in real-world software development; the final judgment is determined by majority vote. We measure usability using the **Usability Rate**, defined as the proportion of target base commits for which at least one generated task instance passes our execution and quality filters across up to five synthesis attempts. An instance must cause at least one previously passing regression test to fail, while its corresponding code changes are also manually judged to be non-trivial rather than simple modifications such as replacing code with `pass`, `TODO`,

[1] `https://anonymous.4open.science/r/SWE-at-scale-FA0A`

Method	Developer	DS-V4-Flash	GPT-5.4-mini	GLM-5.3-Flash	Qwen3.8-Flash
Real-world Tasks					
DeepSWE	Human	54.40	24.30	63.40	58.70
SWE-bench Verified	Human	91.80	69.60	94.00	97.00
Synthetic Tasks					
LLM-Modify	DS-V4-Flash	79.85	35.82	57.78	63.96
	GPT-5.4-mini	75.17	56.04	59.06	66.78
	Qwen3.5-9B	77.84	40.21	51.55	64.86
LLM-Rewrite	DS-V4-Flash	67.80	25.14	47.36	56.32
	GPT-5.4-mini	60.66	31.86	45.15	55.93
	Qwen3.5-9B	70.90	38.52	48.36	72.84
FeatAdd	DS-V4-Flash	58.62	37.93	62.07	62.07
SWE-at-Scale		48.48	31.68	42.15	44.44
SWE-at-Scale+		**33.19**	**16.81**	**35.32**	**37.02**

Table 1: Downstream issue resolution performance (%) across real-world and synthesized SWE task instances. *Developer* denotes the model or human source that constructed each task instance.

or `return None`. Finally, we use **Pass@1** to measure task difficulty based on the success rate of downstream issue resolution.

4 Results

4.1 Synthesis Quality and Task Difficulty

SWE-at-Scale synthesizes more usable and realistic SWE tasks. Figure 5 compares the Usability Rate and HPR of different synthesis methods. Both SWE-at-Scale variants occupy a more favorable region of the usability–realism space than existing synthetic baselines. SWE-at-Scale+ achieves the strongest joint performance, with a 94.0% Usability Rate and a 61.2% HPR, while SWE-at-Scale achieves 72.6% and 55.9%, respectively. In comparison, the best LLM-Rewrite configuration reaches 72.2% usability and 51.8% HPR, while LLM-Modify exhibits substantially lower human preference across generation models. The gap between the two baselines is consistent with their construction procedures. LLM-Modify explicitly prompts an LLM to edit selected code locations to inject an issue, which can produce localized mutations with little connection to how defects arise during ordinary development. LLM-Rewrite instead asks the model to reimplement existing code entities, so an incorrect implementation can emerge as an unintended behavioral regression, a process that more closely resembles the accidental introduction of bugs by human engineers. FeatAdd similarly achieves high realism by deriving bugs from regressions that arise during feature development; however, the open-ended nature of feature implementation provides weak control over whether a usable regression will actually emerge, resulting in substantially lower bug-inducing usability. We attribute the stronger performance of SWE-at-Scale+ to its explicit representation of bug logic in natural language. Rather than requiring the model to implicitly infer bug-inducing logic from code transformations, SWE-at-Scale+ externalizes this logic as a natural-language bug pattern, providing a more explicit learning signal that aligns well with the language-modeling capabilities of LLMs. These results suggest that learning bug-inducing transformations by reversing real-fix transformations captures bug patterns that better reflect real development practice than mechanisms that explicitly prompt an LLM to modify or rewrite code to naturally inject an issue.

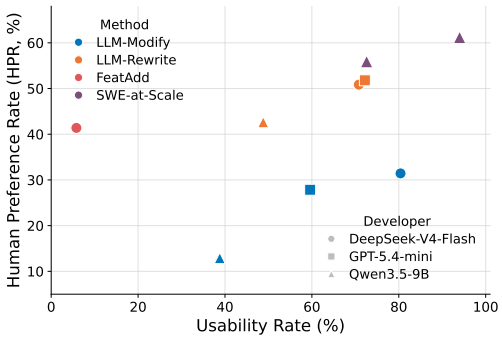


Figure 5: Usability Rate versus Human Preference Rate (HPR).

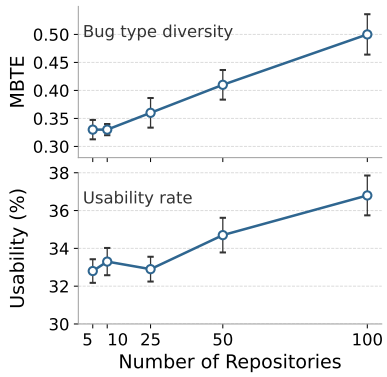


Figure 6: Effect of repository coverage on bug type diversity and usability.

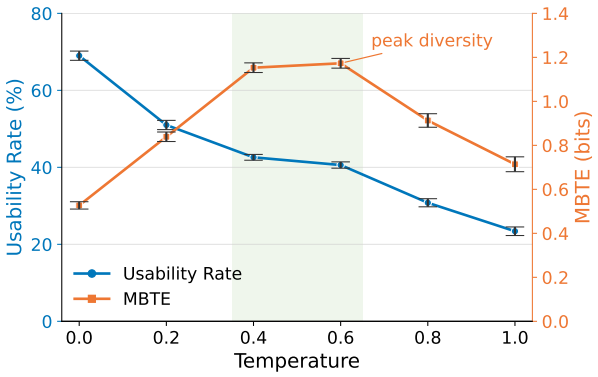


Figure 7: Effect of sampling temperature on bug type diversity and usability.

SWE-at-Scale produces synthetic bugs whose difficulty is competitive with human-expert-designed SWE tasks. As shown in Table 1, both SWE-at-Scale variants consistently yield lower Pass@1 than SWE-bench Verified and existing synthetic baselines across the evaluated resolution models, with SWE-at-Scale+ achieving the lowest Pass@1 in all settings. This consistent trend indicates that the observed difficulty generalizes across different resolution models rather than being specific to a particular solver. Notably, SWE-at-Scale-generated tasks also approach the difficulty of DeepSWE, whose tasks are designed by human experts. This result provides further evidence that supervision derived from reversed real-fix transformations can teach models to construct challenging SWE tasks that approach the difficulty of expert-designed ones.

4.2 Scaling Bug Diversity at Training and Inference Time

We study three complementary factors that shape the diversity of SWE-at-Scale-generated bugs: **repository coverage**, **sampling temperature**, and **sampling budget**. Repository coverage determines the diversity of bug-fixing patterns exposed to the model during training. At inference time, sampling temperature controls how broadly the model explores the learned bug-inducing distribution for a given code context, while sampling budget determines how extensively that distribution is sampled. Together, these factors allow us to examine how bug diversity scales through both training-time data coverage and inference-time sampling.

Mean Bug Type Entropy. We measure how many different types of bugs SWE-at-Scale can generate for a single file. This differs from counting bug types over the whole dataset, where diversity can come simply from using many different files. To this end, we treat each target file as a fixed code context and quantify the diversity of its generated candidates using *Mean Bug Type Entropy* (MBTE) (Chen et al., 2026a; Garg et al., 2026). For target file i under setting s, we compute its entropy as $H_{i,s} = -\sum_k p_{i,s}(k) \log_2 p_{i,s}(k)$, where $p_{i,s}(k)$ denotes the proportion of usable candidates assigned to bug type k. We then macro-average across the N target files and report $\mathrm{MBTE}_s = \frac{1}{N} \sum_{i=1}^{N} H_{i,s}$. Higher MBTE therefore indicates that, on average, the model can generate a broader distribution of bug types from the same code context.

Broader repository coverage enables SWE-at-Scale to learn a more diverse bug distribution. To isolate the effect of repository coverage from training-set size, we train otherwise identical models using the same 2,000 training datapoints while varying only the number of repositories represented in the training data. For each model, we randomly select $N = 50$ target files spanning all 12 repositories in SWE-bench Verified and generate ten independent candidates for each file with sampling temperature $t = 0$, yielding 500 candidates per model. As shown in Figure 6, MBTE increases as the training pool expands from 5 to 100 repositories, while usability remains relatively stable and improves modestly for the largest training pools. This result shows that both the diversity and usability of synthesized bugs increase as the number of training repositories grows.

Moderate sampling temperatures maximize within-file bug diversity, while further increasing the temperature reduces both diversity and usability. We randomly select $N = 50$ target files

spanning all 12 repositories in SWE-bench Verified and generate ten independent candidates for each file at six temperatures, $\{0.0, 0.2, 0.4, 0.6, 0.8, 1.0\}$, yielding 3,000 candidates in total.

As shown in Figure 7, MBTE increases from 0.527 at temperature 0.0 to 1.153 at 0.4 and peaks at 1.173 at 0.6, demonstrating that sampling enables the same target file to yield different bug types. However, usability decreases from 69.0% at temperature 0.0 to 23.4% at 1.0, while entropy also declines to 0.714 at the highest temperature. Moderate temperatures of 0.4–0.6 therefore provide the most favorable diversity–usability trade-off: they substantially expand the range of bugs generated from the same code context, whereas further increasing temperature primarily sacrifices usability without introducing additional diversity.

Larger sampling budgets continuously expand bug diversity while preserving synthesis usability. To study inference-time scaling, we fix both the target files and sampling temperature and progressively increase the sampling budget. Specifically, we use the same 50 target files spanning 12 repositories, set the temperature to $t = 0.6$, and evaluate nested budgets of $K \in \{5, 25, 50, 100, 200\}$ using prefixes of the same 205-sample pool. As shown in Figure 8, SWE-at-Scale's MBTE increases from 0.583 at $K = 5$ to 2.063 bits at $K = 200$, while usability stabilizes at approximately 47% beyond $K = 50$. MBTE grows rapidly at smaller sampling budgets and begins to converge around $K = 100$. Nevertheless, it continues to increase steadily up to $K = 200$, indicating that additional sampling still expands bug-type diversity rather than reaching complete saturation. At the bug-patch level, 4,192 of 4,749 usable candidates (88.27%) remain unique after normalized-diff deduplication at $K = 200$.

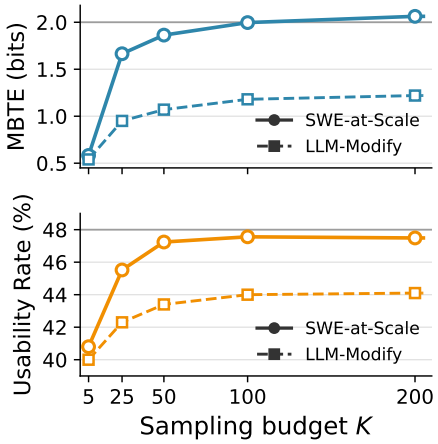


Figure 8: Sampling-budget scaling for SWE-at-Scale and LLM-Modify.

In comparison, LLM-Modify with DS-V4-Flash reaches only 1.22 bits of MBTE and 44.1% usability at $K = 200$, with 61.74% of its usable candidates remaining unique at the normalized-patch level. The substantially higher MBTE and unique-patch rate of SWE-at-Scale suggest that explicit training on reversed real-fix transformations enables the model to explore a broader space of bug types and concrete realizations as the sampling budget grows, whereas a general LLM without dedicated bug-inducing training exhibits substantially stronger repetition.

4.3 Bug Complexity and Synthesis Cost

SWE-at-Scale synthesizes bugs that require larger code changes. As shown in Appendix Figure 9a, SWE-at-Scale and SWE-at-Scale+ modify 18.793 and 18.536 lines on average, respectively, compared with 14.334 lines in SWE-bench Verified. The larger bug-inducing patches suggest that SWE-at-Scale captures more complex code changes rather than being limited to small local perturbations. We exclude LLM-Rewrite from this comparison because its rewritten regions contain substantial code that reconstructs the original functionality and therefore do not provide a comparable measure of bug-inducing changes. Figure 9b shows that the direct SWE-at-Scale path consumes 17.45k tokens per instance on average, more than LLM-Modify (4.46–10.23k) but less than LLM-Rewrite (34.92–62.82k). FeatAdd is an extreme outlier at 2.29M tokens per instance, over two orders of magnitude above SWE-at-Scale. Thus, SWE-at-Scale synthesizes relatively large bug-inducing patches while maintaining substantially lower inference cost than rewrite-based synthesis.

5 Related Work

5.1 Real-World Software Engineering Benchmarks

Repository-level software engineering evaluation (Chen et al., 2026d) has rapidly evolved around SWE-bench, which transforms real-world GitHub issues into executable task instances for evaluating coding agents (Chang et al., 2026; Zeng et al., 2025; 2026a;b; Gao et al., 2026; Hu et al.,

2026; Chen et al., 2024; 2025a; Shi et al., 2025; Wang et al., 2026; Zhang et al., 2026; Pan et al., 2025a). SWE-bench (Jimenez et al., 2024) established repository-level issue resolution as an executable benchmark by pairing 2,294 real GitHub issues with their codebases and regression tests. SWE-rebench (Badertdinov et al., 2025) develops an automated pipeline that continuously extracts interactive tasks from GitHub and supports fresh, repository-disjoint evaluation to reduce contamination. SWE-Bench Pro (Deng et al., 2025) targets substantially longer-horizon, enterprise-level work through 1,865 human-verified tasks drawn from public, held-out, and commercial repositories. DeepSWE (Huang et al., 2026) introduces 113 original, long-horizon SWE tasks across 91 open-source repositories and five programming languages, with newly authored reference solutions that are not merged into upstream repositories to reduce contamination from public development histories. SWE-Bench ProMax (Shi et al., 2026c) instead evaluates large-scale, behavior-preserving refactoring with 170 expert-curated instances derived from real commits across seven languages.

However, naturally occurring task collections remain bounded by the pace of software development, repository availability, and the cost of task authoring and validation. SWE-at-Scale complements these efforts by learning from real software fixes to synthesize new executable task instances beyond the observed set of real-world issues.

5.2 Synthetic Software Engineering Tasks

To overcome the limited scale and coverage of naturally occurring SWE tasks (Lyu et al., 2026; Chen et al., 2026e; Sun et al., 2026), recent work (Jain et al., 2025; Pan et al., 2025b; Lee et al., 2025) has explored scalable ways to construct executable software engineering tasks. SWE-smith (Yang et al., 2025) automatically constructs executable environments for Python repositories and applies multiple bug-generation procedures, including LLM-based rewriting of code entities from their specifications rather than modifying their original implementations, thereby introducing implementation-level discrepancies that can naturally arise during software development. In contrast, SWE-Synth (Pham et al., 2025) performs targeted LLM-based rewriting of existing code, followed by execution-based filtering, to generate verifiable bug-fix pairs and their intermediate repair trajectories. BugPilot (Sonwane et al., 2025) asks an agent to implement a feature and retains unintended test regressions as synthetic bugs, making bug creation resemble feature development rather than explicit mutation. SWE-Mirror (Wang et al., 2025) abstracts the behavioral semantics of an existing issue and re-instantiates the same problem pattern in another functionally similar repository with a configured execution environment.

Existing methods largely rely on inference-time prompting or source-issue transfer, so their synthesis diversity is constrained by the bug-generation knowledge already encoded in general LLMs or by the specific issues used as templates. In contrast, SWE-at-Scale converts real software fixes into explicit training supervision and learns a reusable bug-inducing distribution. This allows the learned bug patterns to be repeatedly instantiated in diverse forms across new code contexts, providing a more scalable way to expand the diversity of synthesized SWE tasks.

6 Discussion

Limitations and future directions. First, our current study focuses exclusively on Python repositories, and it remains unclear whether the learned reverse-transformation paradigm generalizes to other programming languages with different syntax, semantics, and software development practices. Consequently, realistic bugs that are not covered by the available test suite may be discarded, limiting the space of tasks that can be constructed from repositories with weak test coverage. Future work could investigate whether bug-inducing models can learn to operate across multiple programming languages, enabling scalable cross-language SWE task synthesis.

Conclusion. This paper presented SWE-at-Scale, a framework for scalable SWE task synthesis that learns bug injection by reversing real software fixes, rather than relying solely on the bug-generation capabilities of general LLMs. Our experiments show that SWE-at-Scale produces realistic and diverse tasks that remain challenging for current issue-resolution models. We further show that its learned bug distribution can be explored at inference time: different bug types can be generated from the same code context, and bug-type diversity continues to increase as the sampling budget grows without fully saturating within 200 generations per file.

AI USE STATEMENT

AI tools were used to assist with grammar checking of the manuscript and with writing code for data processing and visualization in the experiments.

REPRODUCIBILITY STATEMENT

Our code is available at https://anonymous.4open.science/r/SWE-at-scale-FA0A.

REFERENCES

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. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/21bec6ace947b1b58967b945c8ac0f10-Abstract-Datasets_and_Benchmarks_Track.html.

Pengyu Chang, Yixiong Fang, Silin Chen, Yuling Shi, Beijun Shen, and Xiaodong Gu. Test vs mutant: Adversarial LLM agents for robust unit test generation. *CoRR*, abs/2602.08146, 2026. doi: 10.48550/ARXIV.2602.08146. URL https://doi.org/10.48550/arXiv.2602.08146.

Guhong Chen, Chenghao Sun, Cheng Fu, Qiyao Wang, Zhihong Huang, Chaopeng Wei, Guangxu Chen, Feiteng Fang, Ahmadreza Argha, Bing Zhao, Xander Xu, Qi Han, Hamid Alinejad-Rokny, Qiang Qu, Binhua Li, Shiwen Ni, Min Yang, H. U. Wei, and Yongbin Li. Beyond quantity: Trajectory diversity scaling for code agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 15676–15691. Association for Computational Linguistics, 2026a. doi: 10.18653/V1/2026.FINDINGS-ACL.768. URL https://doi.org/10.18653/v1/2026.findings-acl.768.

Junkai Chen, Xing Hu, Zhenhao Li, Cuiyun Gao, Xin Xia, and David Lo. Code search is all you need? improving code suggestions with code search. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3639085. URL https://doi.org/10.1145/3597503.3639085.

Junkai Chen, Zhiyuan Pan, Xing Hu, Zhenhao Li, Ge Li, and Xin Xia. Reasoning runtime behavior of a program with llm: How far are we? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 1869–1881. IEEE Press, 2025a. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00012. URL https://doi.org/10.1109/ICSE55347.2025.00012.

Silin Chen, Shaoxin Lin, Xiaodong Gu, Yuling Shi, Heng Lian, Longfei Yun, Dong Chen, Weiguo Sun, Lin Cao, and Qianxiang Wang. Swe-exp: Experience-driven software issue resolution. *CoRR*, abs/2507.23361, 2025b. doi: 10.48550/ARXIV.2507.23361. URL https://doi.org/10.48550/arXiv.2507.23361.

Silin Chen, Han Li, Xiaodong Gu, Yuling Shi, and Haibing Guan. Skillforge: Self-distilling agents for project-specific issue resolution, 2026b. URL https://arxiv.org/abs/2608.18933.

Silin Chen, Haoyi Teng, Xiaodong Gu, Yuling Shi, Jiale Huang, Yongpan Wang, Hongyu Zhang, and Haibing Guan. Repo0: Design-driven zero-to-all code generation, 2026c. URL https://arxiv.org/abs/2608.19854.

Zhi Chen, Zhensu Sun, Yuling Shi, David Lo, and Lingxiao Jiang. Are performance-optimization benchmarks reliably measuring coding agents? *CoRR*, abs/2607.01211, 2026d. doi: 10.48550/ARXIV.2607.01211. URL https://doi.org/10.48550/arXiv.2607.01211.

Zhi Chen, Zhensu Sun, Yuling Shi, Chao Peng, Xiaodong Gu, David Lo, and Lingxiao Jiang. Rethinking the value of agent-generated tests for llm-based software engineering agents. *CoRR*, abs/2602.07900, 2026e. doi: 10.48550/ARXIV.2602.07900. URL https://doi.org/10.48550/arXiv.2602.07900.

DeepSeek-AI. Deepseek-v4: Towards highly efficient million-token context intelligence. *CoRR*, abs/2606.19348, 2026. doi: 10.48550/ARXIV.2606.19348. URL https://doi.org/10.48550/arXiv.2606.19348.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, Karmini Sampath, Maya Krishnan, Srivatsa Kundurthy, Sean Hendryx, Zifan Wang, Chen Bo Calvin Zhang, Noah Jacobson, Bing Liu, and Brad Kenstler. Swe-bench pro: Can AI agents solve long-horizon software engineering tasks? *CoRR*, abs/2509.16941, 2025. doi: 10.48550/ARXIV.2509.16941. URL https://doi.org/10.48550/arXiv.2509.16941.

Shuzheng Gao, Wenhao Zeng, Zhaojian Yu, Jianqiao Wangni, Chaozheng Wang, Kai Cai, Shilin He, and Michael R. Lyu. Swe-mem: Learning adaptive memory management for long-horizon coding agents. *CoRR*, abs/2606.28434, 2026. doi: 10.48550/ARXIV.2606.28434. URL https://doi.org/10.48550/arXiv.2606.28434.

Ishir Garg, Neel Kolhe, Xuandong Zhao, and Dawn Song. Infosynth: Information-guided benchmark synthesis for llms. *CoRR*, abs/2601.00575, 2026. doi: 10.48550/ARXIV.2601.00575. URL https://doi.org/10.48550/arXiv.2601.00575.

GLM. GLM-5: from vibe coding to agentic engineering. *CoRR*, abs/2602.15763, 2026. doi: 10.48550/ARXIV.2602.15763. URL https://doi.org/10.48550/arXiv.2602.15763.

Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–22, 2025.

Chao Hu, Wenhao Zeng, Yuling Shi, Beijun Shen, and Xiaodong Gu. In line with context: Repository-level code generation via context inlining. *Proc. ACM Softw. Eng.*, 3(FSE):1469–1491, 2026. doi: 10.1145/3797094. URL https://doi.org/10.1145/3797094.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Wenqi Huang, Charley Lee, Leonard Tng, and Serena Ge. Deepswe: Measuring frontier coding agents on original, long-horizon engineering tasks. *CoRR*, abs/2607.07946, 2026. doi: 10.48550/ARXIV.2607.07946. URL https://doi.org/10.48550/arXiv.2607.07946.

Naman Jain, Jaskirat Singh, Manish Shetty, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environments and hybrid verifiers for scaling open-weights SWE agents. *CoRR*, abs/2504.07164, 2025. doi: 10.48550/ARXIV.2504.07164. URL https://doi.org/10.48550/arXiv.2504.07164.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Hyunji Lee, Minseon Kim, Chinmay Singh, Matheus Pereira, Atharv Sonwane, Isadora White, Elias Stengel-Eskin, Mohit Bansal, Zhengyan Shi, Alessandro Sordoni, Marc-Alexandre Côté, Xingdi Yuan, and Lucas Caccia. Gistify! codebase-level understanding via runtime execution. *CoRR*, abs/2510.26790, 2025. doi: 10.48550/ARXIV.2510.26790. URL https://doi.org/10.48550/arXiv.2510.26790.

Haotian Lin, Silin Chen, Xiaodong Gu, Yuling Shi, Chengxi Pan, Jiaqi Ge, Mengfan Li, Jianghong Huang, Mengchieh Chuang, Beijun Shen, and Haibing Guan. Know before fix: Qa-driven repository knowledge acquisition for software issue resolution. *CoRR*, abs/2607.11111, 2026. doi: 10.48550/ARXIV.2607.11111. URL https://doi.org/10.48550/arXiv.2607.11111.

Yalan Lin, Yingwei Ma, Rongyu Cao, Binhua Li, Fei Huang, Xiaodong Gu, and Yongbin Li. Llms as continuous learners: Improving the reproduction of defective code in software issues. *CoRR*, abs/2411.13941, 2024. doi: 10.48550/ARXIV.2411.13941. URL https://doi.org/10.48550/arXiv.2411.13941.

Yunbo Lyu, David Williams, Jieke Shi, Zhensu Sun, Chao Peng, Zhou Yang, Federica Sarro, and David Lo. How do practitioners build SE agents? insights from a mixed-methods study. *CoRR*, abs/2607.10856, 2026. doi: 10.48550/ARXIV.2607.10856. URL https://doi.org/10.48550/arXiv.2607.10856.

Dongjian Ma, Silin Chen, Yufei Yang, Yuling Shi, Yanfu Yan, and Xiaodong Gu. LLM agents can see code repositories. *CoRR*, abs/2606.14061, 2026. doi: 10.48550/ARXIV.2606.14061. URL https://doi.org/10.48550/arXiv.2606.14061.

Dangfeng Pan, Zhensu Sun, Cenyuan Zhang, David Lo, and Xiaoning Du. The hidden cost of readability: How code formatting silently consumes your LLM budget. *CoRR*, abs/2508.13666, 2025a. doi: 10.48550/ARXIV.2508.13666. URL https://doi.org/10.48550/arXiv.2508.13666.

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 Aarti Singh, Maryam Fazel, Daniel Hsu, Simon Lacoste-Julien, Felix Berkenkamp, Tegan Maharaj, Kiri Wagstaff, and Jerry Zhu (eds.), *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*, volume 267 of *Proceedings of Machine Learning Research*. PMLR / OpenReview.net, 2025b. URL https://proceedings.mlr.press/v267/pan25g.html.

Minh V. T. Pham, Huy Nhat Phan, Hoang Nhat Phan, Cuong Chi Le, Tien N. Nguyen, and Nghi D. Q. Bui. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *CoRR*, abs/2504.14757, 2025. doi: 10.48550/ARXIV.2504.14757. URL https://doi.org/10.48550/arXiv.2504.14757.

Zihan Qiu, Zekun Wang, Xiao Li, Yanpeng Li, Yang Xu, Yixuan Wang, Huaqing Zhang, Rui Men, Bochao Mao, Chengruidong Zhang, Fan Zhou, Hao Luo, Haofeng Huang, Haoran Lian, Haoyan Huang, Hongqing Chen, Jianwei Zhang, Jing Xu, Junjie Wang, Langshi Chen, Liangyu Wang, Linlang Jiang, Man Yuan, Minmin Sun, Peng Jin, Siqi Zhang, Siyu Wang, Xingzhang Ren, Yakai Wang, Yi Zhang, Yiming Dong, Yizhong Cao, Yubo Ma, Yunfei Mao, Bo Zheng, and Dayiheng Liu. On the design of qwen3.8-next architecture: Evaluation, efficiency, and training stability. *CoRR*, abs/2608.30320, 2026. doi: 10.48550/ARXIV.2608.30320. URL https://doi.org/10.48550/arXiv.2608.30320.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Yuchen Shao, Yuheng Huang, Jiawei Shen, Lei Ma, Ting Su, and Chengcheng Wan. Are llms correctly integrated into software systems? In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, ICSE '25, pp. 1178–1190. IEEE Press, 2025. ISBN 9798331505691. doi: 10.1109/ICSE55347.2025.00204. URL https://doi.org/10.1109/ICSE55347.2025.00204.

Yuling Shi, Yichun Qian, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. Longcodezip: Compress long context for code language models. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 141–153. IEEE Press, 2025. doi: 10.1109/ASE63991.2025.00020. URL https://doi.org/10.1109/ASE63991.2025.00020.

Yuling Shi, Zhensu Sun, Junsen Dong, Chengcheng Wan, David Lo, and Xiaodong Gu. Earlyeval: Cheaper agent evaluation via early outcome prediction, 2026a. URL https://arxiv.org/abs/2609.02783.

Yuling Shi, Chaoxiang Xie, Zhensu Sun, Yeheng Chen, Chenxu Zhang, Longfei Yun, Chengcheng Wan, Hongyu Zhang, David Lo, and Xiaodong Gu. Seeing is coding: On the effectiveness of vision language models in code understanding, 2026b. URL https://arxiv.org/abs/2602.01785.

Yuling Shi, Jinghan Xu, Kelin Fu, Wenhao Zeng, Shilin He, Lei Zhang, Yue Liu, Zelin Zhao, Terry Yue Zhuo, Jialun Cao, Siyu Ye, Tianyu Liu, Kai Cai, Shing-Chi Cheung, and Xiaodong Gu. Swe-bench promax: Benchmarking agents on large-scale multilingual code refactoring, 2026c. URL https://arxiv.org/abs/2608.09802.

Atharv Sonwane, Isadora White, Hyunji Lee, Matheus Pereira, Lucas Caccia, Minseon Kim, Zhengyan Shi, Chinmay Singh, Alessandro Sordoni, Marc-Alexandre Côté, and Xingdi Yuan. Bugpilot: Complex bug generation for efficient learning of SWE skills. *CoRR*, abs/2510.19898, 2025. doi: 10.48550/ARXIV.2510.19898. URL https://doi.org/10.48550/arXiv.2510.19898.

Zhensu Sun, Zhihao Lin, Zhi Chen, Chengran Yang, Mingyi Zhou, Li Li, and David Lo. Executing as you generate: Hiding execution latency in llm code interpreters, 2026. URL https://arxiv.org/abs/2604.00491.

Junhao Wang, Daoguang Zan, Shulin Xin, Siyao Liu, Yurong Wu, and Kai Shen. Swe-mirror: Scaling issue-resolving datasets by mirroring issues across repositories. *CoRR*, abs/2509.08724, 2025. doi: 10.48550/ARXIV.2509.08724. URL https://doi.org/10.48550/arXiv.2509.08724.

Yuhang Wang, Yuling Shi, Mo Yang, Rongrui Zhang, Shilin He, Heng Lian, Yuting Chen, Siyu Ye, Kai Cai, and Xiaodong Gu. Swe-pruner: Self-adaptive context pruning for coding agents. *CoRR*, abs/2601.16746, 2026. doi: 10.48550/ARXIV.2601.16746. URL https://doi.org/10.48550/arXiv.2601.16746.

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics, ACL 2025, Vienna, Austria, July 27 - August 1, 2025*, volume ACL 2025 of *Findings of ACL*, pp. 1123–1139. Association for Computational Linguistics, 2025. doi: 10.18653/V1/2025.FINDINGS-ACL.62. URL https://doi.org/10.18653/v1/2025.findings-acl.62.

John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang (eds.), *Advances in Neural Information Processing Systems*, volume 37, pp. 50528–50652. Curran Associates, Inc., 2024. doi: 10.52202/079017-1601. URL https://proceedings.neurips.cc/paper_files/paper/2024/file/5a7c947568c1b1328ccc5230172e1e7c-Paper-Conference.pdf.

John Yang, Kilian Lieret, Carlos E. Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025,*

San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/8b86cf5ace600c48fd188efbb8dedec8-Abstract-Datasets_and_Benchmarks_Track.html.

Wenhao Zeng, Yaoning Wang, Chao Hu, Yuling Shi, Chengcheng Wan, Hongyu Zhang, and Xiaodong Gu. Pruning the unsurprising: Efficient llm reasoning via first-token surprisal. *arXiv preprint arXiv:2508.05988*, 2025.

Wenhao Zeng, Yuling Shi, Xiaodong Gu, Chao Hu, Chaofan Wang, Yuhao Cui, Hongting Zhou, Mengnan Qi, Jianqiao Wangni, Zhaojian Yu, Shuzheng Gao, Kai Cai, and Shilin He. Dockerless: Environment-free program verifier for coding agents. *CoRR*, abs/2606.28436, 2026a. doi: 10.48550/ARXIV.2606.28436. URL https://doi.org/10.48550/arXiv.2606.28436.

Wenhao Zeng, Xuteng Zhang, Yuling Shi, Chao Hu, Yuting Chen, Beijun Shen, and Xiaodong Gu. Glimprouter: Efficient collaborative inference by glimpsing one token of thoughts. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics, ACL 2026, San Diego, California, United States, July 2-7, 2026*, pp. 17850–17864. Association for Computational Linguistics, 2026b. doi: 10.18653/V1/2026.FINDINGS-ACL.885. URL https://doi.org/10.18653/v1/2026.findings-acl.885.

Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/d83c4a745789690f82e86d0ef752ae7c-Abstract-Datasets_and_Benchmarks_Track.html.

Xuteng Zhang, Wenhao Zeng, Xiaodong Gu, Chao Hu, Haotian Lin, Yuling Shi, Min Wang, and Beijun Shen. Paratempo: Efficient parallel reasoning via temporal confidence. *arXiv preprint arXiv:2608.16425*, 2026.

APPENDIX

A.1 LORA FINE-TUNING CONFIGURATION

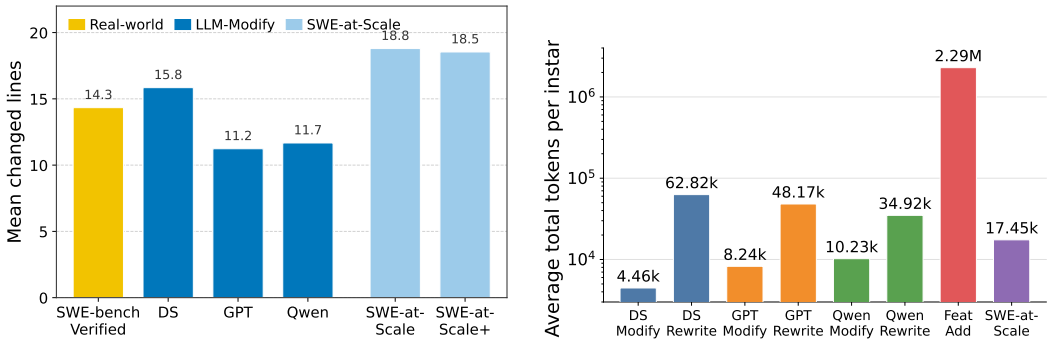
We report the complete LoRA fine-tuning configuration used to train both SWE-at-Scale and SWE-at-Scale+. Table 2 summarizes the backbone, optimization hyperparameters, and training settings used in all experiments.

Hyperparameter	Value
Base model	Qwen3.5-9B
Fine-tuning method	LoRA supervised fine-tuning
LoRA rank r	16
LoRA scaling α	32
LoRA dropout	0.05
Target modules	All linear layers
Training examples	13,407
Maximum sequence length	64,000 tokens
Number of epochs	1
Per-device batch size	1
Gradient accumulation steps	8
Effective batch size	8
Learning rate	1×10^{-4}
Learning-rate scheduler	Cosine
Warmup ratio	0.03
Weight decay	0
Maximum gradient norm	1.0
Numerical precision	bfloat16
Gradient checkpointing	Enabled

Table 2: LoRA fine-tuning configuration used for both SWE-at-Scale and SWE-at-Scale+.

A.2 BUG COMPLEXITY AND SYNTHESIS COST

Figure 9 reports the mean number of changed lines per synthesized instance and the average total token consumption required to produce one instance.



(a) Mean changed lines per instance.

(b) Average total tokens per instance.

Figure 9: Mean changed lines and average total token consumption across the evaluated synthesis configurations.