BugPilot: Complex Bug Generation for Efficient Learning of SWE Skills

Anonymous authors
Paper under double-blind review

Abstract

Training the next generation of language model based software engineering (SWE) agents requires bug-fixing data sourced from high-quality bugs. We introduce a novel method for the synthetic generation of difficult and diverse bugs. Our method instructs SWE Agents to introduce a feature into the codebase and collects *unintentionally* buggy changes caught by test failures. Prior approaches focus on generating bugs *intentionally* (e.g., local perturbation to existing code) and are unreflective of realistic development processes, yielding out-of-distribution bugs. Qualitative analysis demonstrates that our approach for generating bugs more closely reflects the patterns found in human-authored edits. Experiments show that our bugs provide more efficient training data for supervised fine-tuning, outperforming models trained on a bug dataset of pre-existing bugs from SWE-Smith and R2E-Gym by over 4%. Finally, training on our newly generated bugs in addition to existing bug datasets results in FrogBoss, a state-of-the-art 32B model on SWE-Bench Verified with a pass@1 of 54.6% averaged over three seeds. Our FrogBoss recipe generalizes across model sizes, demonstrated by FrogMini and FrogBeast, with 14B and 235B parameters with SWE-Bench Verified pass@1 of 45.3% and 61.0% respectively.

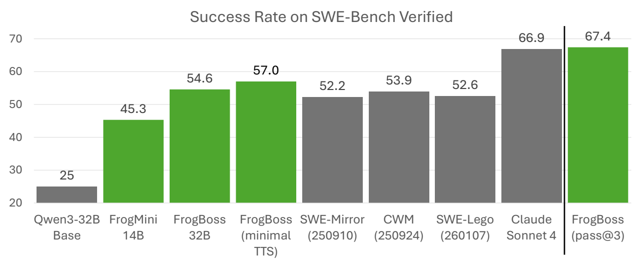

Figure 1: **Comparison to previous SoTA results.** We train FrogBoss with our collected data including our new FeatAdd datasets, it achieves 54.6% pass@1 averaged over three seeds. With pass@3 we achieve a score of 67.4%, outperforming Claude Sonnet 4, illustrating the performance gains we could get with a good verifier. The minimal test time scaling refers to the strategy of selecting the shortest trajectory among three rollouts.

1 Introduction

Large language model (LLM)–based agents have recently made strong progress on software engineering (SWE) tasks (Yang et al., 2024; Jimenez et al., 2024; Pan et al., 2025; Wei et al., 2025). However, the strongest agents rely on proprietary models, and improving open-weight models on these tasks remains challenging. Training models to solve bugs using either supervised fine-tuning from expert data or reinforcement learning is promising (Yang et al., 2025b; Jain et al., 2025; Luo et al., 2025; Wei et al., 2025). However scaling this approach requires large, high-quality bug datasets.

Existing bug curation strategies fall into two camps. One mines real bugs from pull requests and commits in open-source repositories, which demands careful issue localisation and filtering (Xie et al., 2025; Badertdinov et al., 2025; Pan et al., 2025; Wang et al., 2025a; Zhang et al., 2025b). Alternatively, synthetic bug generation injects faults into existing codebases, allowing researchers

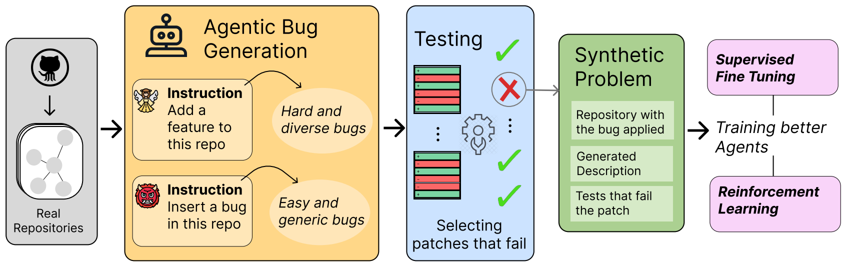

Figure 2: **Illustration of our BugPilot pipeline.** First, we instruct SWE Agent (Yang et al., 2024) with Claude Sonnet 4 to introduce bugs, either through deliberate attempts (BugInstruct) or by attempting to add a feature (FeatAdd). Then, we check whether these modifications resulted in the tests for the repository failing. If the tests fail, then we add this to our dataset of bugs.

to scale data without being bottlenecked by the availability of existing commits, pull requests or issues (Yang et al., 2025b). A notable example of synthetic bugs is SWE-Smith (Yang et al., 2025b), which relies on hand-engineered rules and LLM re-implementations of existing functions in order to perturb the codebase until tests break. Although useful, this method produces datasets skewed toward a narrow set of bug types with fixes that are short and typically confined to a single file. This might undermine the transferability of models trained on such synthetic data to real-world scenarios, where bugs typically arise through natural development processes rather than deliberate injection of errors.

In this work, we introduce BugPilot, a novel approach to synthetic bug generation that leverages software engineering agents to create more naturalistic bugs through realistic development workflows. A naive approach to agentic bug generation would be what we refer to as BugInstruct: to explicitly instruct a SWE agent to *intentionally* introduce a bug in an existing code-base. This approach generates bugs that qualitatively do not resemble realistic bugs. Rather than intentionally injecting errors, our method tasks SWE agents with developing new features within existing repositories (which we refer to as FeatAdd). This results in naturally introduced bugs when these implementations *unintentionally* break existing test suites. We detect when such breakages arise and record the state of the repository at this point as containing a bug that needs to be resolved. This process mirrors authentic software development scenarios where bugs commonly arise as unintended side effects of feature development and code modifications.

Through qualitative and quantitative analyses, we demonstrate that our generated bugs are not only more challenging for current agents, but are more diverse and exhibit more natural characteristics compared to existing synthetic datasets. Comparing *unintentionally* generated bugs (FeatAdd) to *intentionally* generated bugs (BugInstruct and SWE-Smith) for fine-tuning of a base model reveals that unintentional bugs provide much more efficient training examples - performing 2% better with half the number of training trajectories (1.2k vs. 2.3k). Using our bugs to train an agent with reinforcement learning, our model achieves 52.4% on SWE-Bench Verified with a 32B parameter model (Pass@1 averaged across 3 seeds). When training using an extended set of all our collected data, we achieve state-of-the-art results for a 32B model with 54.6% on SWE-Bench Verified.

Our contributions are fourfold: (1) we propose BugPilot, a novel methodology for generating synthetic bugs through realistic development workflows with SWE agents (Figure 2), (2) through qualitative analysis we categorise bug datasets and show that bugs generated through FeatAdd reflect a more natural category distribution (Section 5.1), (3) we demonstrate that *unintentionally* generated bugs provide more efficient training data for supervised fine-tuning and reinforcement learning than *intentionally* generated bugs, (4) we train FrogBoss , a 32B model with 54.6% Pass@1 averaged over three seeds and FrogMini, a 14B model with 45.3% averaged over three seeds by training with on a dataset combining all of our unintentionally generated FeatAdd bugs with previous bug datasets.

2 Related Work

Software Engineering Benchmarks and Tasks. SWE-Bench (Jimenez et al., 2024) was one of the first benchmarks for evaluating LLM agents at resolving bugs at a repository level from natural language bug reports, introducing a set of $2,294$ problems. SWE-Bench Verified further curated a subset verified by human engineers as solvable given the problem statement, addressing issues of underspecification with the original dataset. Subsequent methods scale the SWE-bench methodology to collect large numbers of *training* tasks from sources such as GitHub: SWE-Fixer (Xie et al., 2025) (110k tasks), SWE Gym (Pan et al., 2025) (2.4k tasks). Scaling to include more (or newer) evaluation tasks often involves automated PR collection pipelines: SWE-rebench (Badertdinov et al., 2025), SWE-bench-Live (Zhang et al., 2025b), SWEE-Bench (Vergopoulos et al., 2025), SWE-Bench++ (Wang et al., 2025b) as well as extracting new kinds of tasks beyond PR resolution (Du et al., 2025). However, data collection from open source repositories is limited by availability of code present on platforms such as GitHub. Recent work has studied the synthetic generation of bugs in the form of SWE-Smith (Yang et al., 2025b), where LLMs rewrite functions and add logical perturbations to add bugs, and R2E-Gym (Jain et al., 2025), which synthesises problems statements to enable bug extraction from the commits rather than GitHub issues. SWE-Synth Pham et al. (2025) proposes generating new test cases and bug-fix pairs by utilizing an LLM to rewrite a given function, while SWE Playground Zhu et al. (2025) synthetically generates entire repositories and tests from scratch to use as training environments.

Frameworks for Software Engineering Agents. A common class of agentic approaches for using LLMs for software tasks involves prompting a language model to interface with a repository with tool calls and edit actions as presented in Yang et al. (2024); Ma et al. (2024); Yuan et al. (2025); Arora et al. (2024); Wang et al. (2025c); Jain et al. (2025). Work shows that implementing new tools such as complex navigation and manipulation tools in Moatless tools (Örwall, 2024) or a pdb debugger in Debug Gym (Yuan et al., 2025) can result in enhanced bug-fixing performance. In contrast to such agentic approaches are pipeline-based approaches such as Agentless (Xia et al., 2024) Aider (Aider-AI, 2025) made up of explicit phases such as localisation or validation as opposed to end-to-end loop agentic loops.

Training SWE Agents. Recent work has seen success in training language models to improve as software engineering agents. Approaches generally collect large amounts of tasks from platforms such as GitHub to use for training via a combination of supervised finetuning and reinforcement learning (Pan et al., 2025; Luo et al., 2025; Wei et al., 2025; Du et al., 2025; Wang & Ammanabrolu, 2025). SWE-Smith Yang et al. (2025b) and SWE-Lego (Tao et al., 2026) rely on using successful trajectories from a more capable teacher model on collected tasks to train smaller models. While EntroPO (Yu et al., 2025) extends existing preference alignment algorithms to multi-turn tool calling environment such as SWE agents. Moreover, approaches such as test-time scaling that use test generation or LLM-as-a-judge yield significant performance improvements on SWE Bench style tasks, using Monte Carlo tree search (MCTS) (Antoniades et al., 2025; Zhang et al., 2025a) or inference time process reward models (Gandhi et al., 2025) to guide better search procedures with verification.

3 Automatic Bug Generation

3.1 Background

A *bug* consists of (1) a repository with some code that functions incorrectly, (2) a natural language description of how this bug manifests while using the code and (3) tests that can be used to verify if the bug has been fixed. In the context of *bug-fixing* tasks, agents are provided with (1) the buggy repository and (2) the problem description and are expected to produce code changes in the repository. These code changes are said to be correctly resolving the bugs if they result in the (3) verification test cases passing.

Collections of bugs, such as in SWE-Bench, serve dual purposes in advancing SWE agents. First, they enable systematic evaluation on realistic debugging scenarios. Second, they provide valuable training data: bugs can be used to generate expert trajectories for supervised fine-tuning, or serve as RL environments for bug-fixing skills. Such bug datasets are traditionally curated from open-source repository histories. However, this approach requires substantial effort to filter suitable examples and is fundamentally constrained by the availability of development histories on platforms like GitHub. Alternatively, synthetic bug generation pipelines enable bug creation for arbitrary repositories and programming languages without depending on existing commit histories. The R2E-Gym dataset constructs bugs by reverting commits from Python repositories and the SWE-Smith dataset introduces synthetic bugs into Python repositories.

Table 1: **Solve statistics of Claude Sonnet 4, GPT-4o and GPT-5 using R2E-Gym as the agentic scaffold.** Agentic bugs generated via either FEATADD or BUGINSTRUCT are more difficult than the SWE-SMITH and R2E-GYM bugs, with FEATADD bugs being the most difficult of them all across the three models.

Models	R2E-GYM	SWE-SMITH	BUGINSTRUCT	FEATADD
Claude Sonnet 4	63.5%	65.9%	54.6%	41.4%
Successful Trajectories	3,208	2,611	2,330	1,243
Avg Steps	42.0	39.2	43.1	45.5
Avg Observation Tokens	460.1	448.8	464.7	433.5
Avg Assistant Content / Trajectory	34.5	30.5	32.9	35.3
Avg Assistant Content Tokens	56.8	59.1	60.4	61.0
GPT-4o	32.8%	29.4%	13.4%	18.5%
Avg Assistant Content / Trajectory	5.6	5.8	6.2	7.7
Avg Assistant Content Tokens	150.7	152.1	157.7	154.4
GPT-5	68.7%	77.5%	67.8%	53.4%
Avg Assistant Content / Trajectory	0.8	0.5	12.3	14.5
Avg Assistant Content Tokens	465.6	481.3	21.0	22.0

3.2 BUGPILOT: AGENTIC GENERATION OF BUGS

Current methods for generating synthetic bugs (e.g. SWE-Smith) work by perturbing the code until the tests break. We hypothesize that SWE agents themselves might be used to introduce bugs in a way more reflective of real-life software engineering. We start with the set of 128 SWE-Smith repositories where, for each repository we have a containerised environment (docker image) where the codebase along with all dependencies have been installed. To synthesize bugs within these repositories we use SWE-Agent (Yang et al., 2024) with Claude Sonnet 4. An illustration of our bug generation pipeline can be found in Figure 2 and a comparison between the bugs generated using FEATADD approach can be found in Figure 5. We consider two ways to introduce bugs within this framework:

Intentional Bug Introduction (BUGINSTRUCT). We instruct the agent to introduce bugs into the repository by enriching its system prompt with guidance on bug integration techniques and verification steps to confirm that changes break existing functionality. However, this intentional approach produces bugs that lack diversity and are typically far simpler than real-world bugs, as we demonstrate later. This distributional mismatch ultimately limits their effectiveness for improving agentic coding performance. We refer to this method as BUGINSTRUCT.

Buggy Feature Addition (FEATADD). Many real world bugs in the software development process arise when existing code is modified incorrectly to support new features. We emulate this by tasking our agent to come up with and implement a new feature for the given repository while preserving existing functionality (detailed prompt provided in Appendix A). Whenever the feature breaks an existing test, a bug is created. Unlike the earlier approach, bugs here are *unintentional* and thus are more likely to align to naturally occuring bugs. We refer to this method as FEATADD.

For both of these approaches, given a repository, we execute both strategies multiple times in order to collect differing approaches from at performing the same task. We evaluate whether a run resulted in a bug by running tests after the agent has submitted and making sure at least one test fails (see Figure 3). The description of the bug for each "buggy" run is generated by prompting the language model to generate a bug-report given the output of the failed tests, following SWE-Smith (Yang et al., 2025b). The final synthetically generated bug is thus composed of the code changes made by the agent during its execution along with the failing test and its outputs. Our approach enjoys the same scalability benefits of SWE-smith, wherein once a repository is setup in a containerised environment, no additional manual effort is required in order to generate more bugs.

4 DATASETS AND TRAINING METHODOLOGY

Agentic Framework. For all our inference and training purposes we use R2EGym as our agentic scaffold, because of its previous usage and strong performance on SWE-Bench Verified (Jain et al., 2025). Moreover, the R2EGym scaffold is built into RLLM (Tan et al., 2025), a framework for

Table 2: **Bug Statistics.** We compare bugs from different generation/collections methods. SWE-Bench Verified contains real-world bugs, R2E-GYM uses human-authored edits, while others use synthetic generated bugs. FEATADD characteristics differ significantly from previous approaches in that the patch used to introduce the bug has many more tokens and on average twice as many files changed.

Feature	SWE-B-V	R2E-GYM	SWE-SMITH	BUGINSTRUCT	FEATADD
Total tasks	500	1000	1000	1000	785
Problem tokens	447.7	264.6	312.0	332.6	304.4
Avg diff patch tokens	394.0	352.6	598.2	435.3	4376.0
Avg files modified	1.2	2.6	1.2	1.3	4.2
Avg net lines changed	5.6	36.0	-3.2	12.4	415.9
Unique repositories	12	10	125	111	86
Avg tasks per repo	41.7	100.0	8.0	9.0	9.1
Claude resolve rate	70.8	63.5	65.9	54.6	41.4

training language models with RL on SWE tasks, making it convenient to compare SFT to RL. The R2EGym scaffold offers to the agent four tools: the `file-editor`, `execute-bash`, `search` and `finish`.

Supervised Fine-tuning. We collect trajectories for training with supervised fine-tuning (SFT) using Claude Sonnet 4 and bug datasets from all four of the bug generation techniques described above. To create the dataset for SFT, we use rejection sampling on each of the datasets. We generate the trajectories using a 64k context length and 10k max prompt length and then filter them based on success. The statistics of these trajectories are reported in Table 1. For our student model, we choose Qwen3-32B (Yang et al., 2025a). We fully fine-tune our model using LlamaFactory (Zheng et al., 2024) with a learning rate of 1e-5, no weight decay, we perform 3 epochs of training with a maximum context length of 32k tokens. If a trajectory generated at 64k tokens is successful, we train on the first 32k tokens. This occupies one node of 8 Nvidia H100 for 54 hours.

Reinforcement Learning. Recent work has shown promise in using Reinforcement Learning to fine-tune LLMs for tasks with verifiable rewards, especially for Maths and Competition programming. Software development tasks as discussed in this paper differ from the above in that they are multi-turn, requiring the model to interact with the environment in a diverse way. Following DeepSWE (Luo et al., 2025), we employ the RLLM (Tan et al., 2025) framework for fine-tuning language models using RL with various rewards. Similar to the SFT paradigm, we generate rollouts with a max context length of 64k tokens, but truncate to the first 32k tokens for training. To train reinforcement learning for 25 steps with 64 bugs per step and 8 rollouts per bug, we require 8 nodes of 8 $\times$ H100s for 50 hours.

Evaluation. Evaluation was performed on SWE-Bench Verified and every model was run over three seeds with a context length of 64k, 100 max steps and a temperature of 1. Full hyper-parameters can be found in Appendix D.

Methodology and Data Mixtures. Our main experiments use a base mixture of R2E-GYM and SWE-SMITH bugs with a total of 5,621 successful resolution trajectories from Claude Sonnet 4 (as reported in Table 1, we have 5,819 trajectories for these two datasets but we leave out 198 trajectories for validation). We call this mixture BASEMIX. We first fine-tune our base model on this mixture followed by performing another round of fine-tuning on 1.2k trajectories generated from our agentic generated BUGINSTRUCT and FEATADD bugs. For a fair comparison, we perform this second stage of fine-tuning on additional 1.2k trajectories (same size as the above datasets) from R2E-GYM and SWE-SMITH, not included in BASEMIX. Finally, we experiment with fine-tuning the base model on ALLDATA, consisting of BASEMIX, FEATADD, along with the 1k trajectories each fromR2E-GYM and SWE-SMITH.

5 RESULTS

5.1 BUG ANALYSIS

Bug Categorization. We first study the characteristics of the bugs generated by our approach, categorizing them and comparing the distribution of bug-types with both human-authored bugs (SWE-Bench Verified and R2E-GYM) and AI-generated (SWE-SMITH). Figure 3 presents results from an

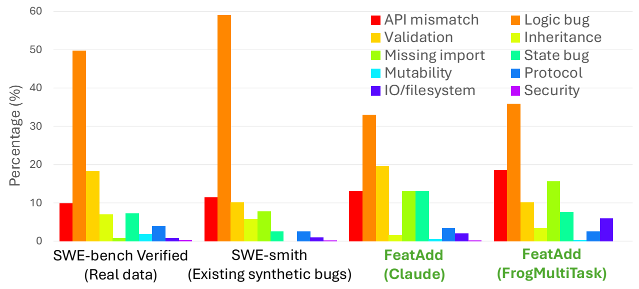

Figure 3: **Distribution of common bug types across different bug datasets.** FEATADD shows the most even distribution of bugs compared to prior work. SWE-SMITH bugs shows a particular skew towards *logic* bugs. This can be explained by the rule-based and local nature of its generation process. The distribution of FEATADD bugs is similar to SWE-Bench, which is closest to the human-authored edit distribution found in real repositories.

LLM-aided categorisation. Bugs generated using FEATADD demonstrate a more even distribution of bugs across various categories compared to prior work, which is skewed to a few bug types. This indicates that compared to prior synthetic generation baselines, the FEATADD approach is more suitable for generating diverse bugs that match real world bug distributions as seen in SWE-Bench Verified. Details of bug categorisation can be found in Appendix C and examples of FEATADD bugs of different types can be found in Appendix B.

Bug Statistics. In Table 2, we study how bug patches differ quantitatively across different generation methods in terms of size of patches in tokens, lines and files. We find that FEATADD results in bug patches spanning multiple files and of a greater magnitude, indicating more complex bugs than other bug generation approaches.

Bug Difficulty. We evaluate bug difficulty by measuring the ability of a strong coding agent to solve generated bugs from that dataset. We use the R2E-Gym *scaffold* (Jain et al., 2025) as our agentic scaffold and we use Claude Sonnet 4 as our strong coding LLM. For all bugs, we sample 4 attempts. The results in Table 1 show that FEATADD bugs are the most challenging even for frontier models with a low resolve rate of 41.4% compared to 63.5% for R2E-GYM and 65.9% for SWE-SMITH.

These results indicate that FEATADD bugs are more diverse, are closer to bugs that might arise from regular development practices as well being as more complex than other bug generation approaches.

5.2 TRAINING ON BUGS

We demonstrate the utility of our bug generation approach by fine-tuning models on generated bugs and comparing with prior work in Table 3. Notably, we train FROGBOSS a state-of-the-art 32B model using a mix of 2k synthetic and real bugs from R2E-GYM, SWE-SMITH and FEATADD datasets. Despite using a 25% smaller dataset of trajectories than previous work (9k vs 12k), FROGBOSS achieves a **54.6%** Pass@1 averaged over three seeds with error bars of 0.67. To demonstrate that our training recipe generalizes across model scales, we apply the FROGBOSS recipe to Qwen3-14B and Qwen3-235B-A22B-Instruct-2507 (a mixture-of-experts model with 22B active parameters), yielding FROGMINI and FROGBEAST with 45.3% and 61.0% Pass@1 respectively (averaged over three seeds).

Effects of including high-quality bug data. Table 4 compares fine-tuning results when training trajectories are drawn from different bug datasets. We find that training with data that includes trajectories drawn from FEATADD bugs results in higher performance: for 32B models, training on BASEMIX (a 5.6k-bug mixture of R2E-GYM and SWE-SMITH) yields **4.73%** lower performance on SWE-Bench Verified than FROGBOSS (trained with data mix including FEATADD). A similar trend holds for 14B models: FROGMINI, trained on a mixture of R2E-GYM, SWE-SMITH, and FEATADD, outperforms our 14B model trained on BASEMIX by **4.14%**. We include BASEMIX as a baseline because R2E-GYM and SWE-SMITH were the most commonly used training bug sources for SWE-agent-style systems prior to this work.

Effectiveness of unintentionally generated bugs. Next, we perform ablations starting from a model trained with BASEMIX and comparing performance when it is further fine-tuned on R2E-GYM,

Table 3: **Comparison to Current State-of-the-Art.** We achieve state-of-the-art results for a 14B and 32B model by supervised fine-tuning on bugs generated by FEATADD in addition to existing bug datasets (ALLDATA). We also show that training with reinforcement learning (RL) on FEATADD for only 25 steps, starting from a base model fine-tuned with BASEMIX, can outperform previous state of the art with less than half the amount of trajectories. Our model trained with supervised fine-tuning (SFT) on BASEMIX + FEATADD achieves near state-of-the-art results with 40% of the total training trajectories (7k vs. 12k) and 5% of the total bugs (3k vs. 60k). The numbers for Devstral 2 with the mini-SWE-Agent scaffold are from the SWE-Bench Verified leaderboard `https://www.swebench.com/`

Model/Method	**Scaffold**	**Bugs**	**Trajectories**	**SWE-Bench Verified**
Proprietary Models				
Claude Sonnet 4	R2E-Gym	-	-	66.9
GPT-5	R2E-Gym	-	-	65.7
Open Weights Models				
Qwen3-Coder-480B (Yang et al., 2025a)	mini-SWE-Agent	-	-	55.4
SWE-Fixer-72B (Xie et al., 2025)	SWE-Fixer	110k	-	32.8
SWE-RL-70B (Wei et al., 2025)	Agentless	-	-	41.0
CWM-32B (FAIR CodeGen Team, 2025)	Agentless	-	-	53.9
DeepSWE-32B-Preview (Luo et al., 2025)	R2E-Gym	4.6k	-	42.2
R2E-Gym-32B (Jain et al., 2025)	R2E-Gym	4.6k	4.5k	34.4
SWE-Smith-LM-32B (Yang et al., 2025b)	SWE-Agent	50k	5k	40.2
SWE-Mirror-LM-32B (Wang et al., 2025a)	OpenHands	60k	12k	52.2
SWE-Lego-32B (Tao et al., 2026)	OpenHands	32k	18k	52.6
Devstral 2-123B (Mistral AI, 2025)	mini-SWE-Agent	-	-	53.8
Devstral Small 2-24B (Mistral AI, 2025)	mini-SWE-Agent	-	-	56.4
Ours (14B)				
BASEMIX + FEATADD 14B (SFT)	R2E-Gym	3k	7k	**41.1**
BASEMIX + FEATADD 14B (RL)	R2E-Gym	3k	5.8k	**42.2**
FROGMINI (All Data, SFT)	R2E-Gym	3k	9k	**45.3**
Ours (32B)				
BASEMIX + FEATADD 32B (SFT)	R2E-Gym	3k	7k	**51.9**
BASEMIX + FEATADD 32B (RL)	R2E-Gym	3k	5.8k	**52.4**
FROGBOSS (All Data, SFT)	R2E-Gym	3k	9k	**54.6**

SWE-SMITH, BUGINSTRUCT, and FEATADD. This setup allows us to isolate the effects of each dataset seperately. From Table 4, we observe that BASEMIX + FEATADD performs 2.1% over BASEMIX + BUGINSTRUCT and 1.4% over BASEMIX + SWE-SMITH. In fact, the inclusion of BUGINSTRUCT into the training mix causes no noticeable improvement over BASEMIX. Both SWE-SMITH and BUGINSTRUCT bugs are *intentionally* generated bugs, while our FEATADD bugs are *unintentionally* generated, highlighting the effectiveness of FEATADD and its importance in the state-of-the-art performance of FROGBOSS.

Reinforcement learning with synthetic bugs. Finally, we ablate training with reinforcement learning. Starting with models trained on BASEMIX, we fine-tune using bugs from FEATADD as environments for RL. From Table 4 we observe that RL leads to improvements over SFT in both the 32B and 14B setting, indicating that it is an effective way of using synthetically generated bugs. However it does not result in better performance than FROGBOSS or FROGMINI. We believe this is due to (1) more quantity and stronger teacher data from Claude Sonnet 4 outweigh the advantages of GRPO, and (2) the lack of entropy and resulting training data diversity after training with SFT.

Impact of Teacher Model. In addition to Claude Sonnet 4, we also attempt to collect agent trajectories using GPT-4o and GPT-5 as LLM backbone. We report the statistics of these trajectories in Table 1. GPT-5 outperforms Claude Sonnet 4 in all four sets of bugs, while GPT-4o struggles to resolve even one third of the bugs, especially on the BUGINSTRUCT and FEATADD sets. However, we observe that the GPT models tend to generate significantly less assistant content in association with the tool/function calls where on R2E-GYM and SWE-SMITH, GPT-5 generates less than one assistant content per trajectory. This leads to substantial differences in performance when using these trajectories for distillation: student models trained on GPT-5 and GPT-4o trajectories result in a success rate of 31.40% and 21.57% on SWE-Bench Verified, respectively, much lower than

Table 4: **Comparison of fine-tuning on different bug datasets.** We report Pass@1 averaged over three seeds, Pass@3, Pass^3 (tasks solved in all three runs), and Pass@Short (best of three with the shortest trajectory).

Model	**Size**	**Pass@1**	**Pass@3**	**Pass^3**	**Pass@Short**
Qwen3-32B	32B	25.00	40.00	12.60	29.40
BASEMIX (SFT)	32B	49.87	63.80	37.00	50.20
BASEMIX + R2E-GYM (SFT)	32B	50.73	63.60	36.60	54.60
BASEMIX + SWE-SMITH (SFT)	32B	50.53	64.60	36.60	52.40
BASEMIX + BUGINSTRUCT (SFT)	32B	49.87	65.00	33.00	49.60
BASEMIX + FEATADD (SFT)	32B	51.93	64.40	39.40	54.60
BASEMIX + FEATADD (RL)	32B	52.40	65.60	38.20	56.80
ALLDATA (FROGBOSS, SFT)	32B	**54.60**	**67.40**	**41.20**	**56.80**
Qwen3-14B	14B	18.33	30.40	8.20	24.40
BASEMIX (SFT)	14B	41.13	54.80	28.40	47.60
BASEMIX + FEATADD (SFT)	14B	40.40	55.40	25.80	45.00
BASEMIX + FEATADD (RL)	14B	42.20	55.00	29.20	45.80
ALLDATA (FROGMINI, SFT)	14B	**45.27**	**58.20**	**32.20**	**49.00**

54.6% on SWE-Bench Verified obtained by training on Claude Sonnet 4 trajectories. Our observation aligns well with recent reasoning curation work (Abdin et al., 2025; Zhao et al., 2025), where they demonstrate that the quality of the reasoning content can be crucial in SFT training in domains such as maths and code generation.

6 DISCUSSION AND FUTURE WORK

Towards Self-Improving Bug Generation. A limitation of our approach is its reliance on a strong teacher model (e.g., Claude Sonnet 4) for bug generation: if future teacher models become sufficiently capable that they no longer inadvertently introduce bugs during feature implementation, the pipeline's effectiveness as a distillation technique could diminish. Additionally, the current FEATADD pipeline is inherently inefficient: by design, we discard all trajectories where the agent successfully implements a feature without breaking tests (precisely the runs that "fail" to produce bugs). This represents a substantial waste of computation, as a significant fraction of agent runs fall into this category.

To begin addressing both concerns, we investigate whether a student model can itself serve as a bug generator while making use of the otherwise-discarded trajectories. We train FROGMULTITASK, a variant that learns from both bug-resolving trajectories (identical to FROGBOSS) and feature-implementation trajectories that did *not* result in test failures. This multi-task objective teaches the model to both resolve bugs and implement new features. Importantly, we find that this joint training does not degrade bug-resolving performance: FROGMULTITASK achieves 54.0% Pass@1 on SWE-Bench Verified (averaged over three seeds), comparable to FROGBOSS. Crucially, the multi-task training unlocks feature-implementation capabilities in the student model, enabling it to serve as a bug generator. As a preliminary investigation, we use FROGMULTITASK to generate 500 new bugs via the FEATADD protocol. As shown in Figure 3, the distribution of bug types produced by FROGMULTITASK closely mirrors that of bugs generated by Claude Sonnet 4, suggesting that the student model can produce similarly diverse and naturalistic bugs.

Iterative Self-Improvement. These preliminary findings point toward a promising direction for future work: an iterative self-improvement loop in which the model alternates between generating bugs and learning to resolve them. After an initial seed iteration using a teacher model (as in our current FEATADD pipeline), subsequent iterations could employ the student model itself to generate new training problems. In such a framework, *all* trajectories would contribute to learning: successful feature implementations improve the model's generative capabilities, while failed implementations (i.e., those that break tests) provide new bugs. Combined with RL training on the generated bugs, this creates a closed loop where the model continuously produces both its own training environments and training signal. We leave systematic exploration of this iterative paradigm, including its scaling properties and potential for continuous improvement, to future work.

References

Marah Abdin, Sahaj Agarwal, Ahmed Awadallah, Vidhisha Balachandran, Harkirat Behl, Lingjiao Chen, Gustavo de Rosa, Suriya Gunasekar, Mojan Javaheripi, Neel Joshi, et al. Phi-4-reasoning technical report. *arXiv preprint arXiv:2504.21318*, 2025.

Aider-AI. Aider: Ai pair programming in your terminal. `https://github.com/Aider-AI/aider`, 2025.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Yang Wang. SWE-search: Enhancing software agents with monte carlo tree search and iterative refinement. In *The Thirteenth International Conference on Learning Representations*, 2025. URL `https://openreview.net/forum?id=G7sIFXugTX`.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.

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 *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2025. URL `https://openreview.net/forum?id=nMpJoVmRy1`.

Yaxin Du, Yuzhu Cai, Yifan Zhou, Cheng Wang, Yu Qian, Xianghe Pang, Qian Liu, Yue Hu, and Siheng Chen. Swe-dev: Evaluating and training autonomous feature-driven software development. *arXiv preprint arXiv:2505.16975*, 2025.

Meta FAIR CodeGen Team. Cwm: An open-weights llm for research on code generation with world models, 2025. URL `https://ai.meta.com/research/publications/cwm/`.

Shubham Gandhi, Jason Tsay, Jatin Ganhotra, Kiran Kate, and Yara Rizk. When agents go astray: Course-correcting swe agents with prms. *arXiv preprint arXiv:2509.02360*, 2025.

Naman Jain, Jaskirat Singh, Manish Shetty, Tianjun Zhang, Liang Zheng, Koushik Sen, and Ion Stoica. R2e-gym: Procedural environment generation and hybrid verifiers for scaling open-weights SWE agents. In *Second Conference on Language Modeling*, 2025. URL `https://openreview.net/forum?id=7evvwwdo3z`.

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

Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, Manan Roongta, Ce Zhang, Li Erran Li, Raluca Ada Popa, Koushik Sen, and Ion Stoica. Deepswe: Training a state-of-the-art coding agent from scratch by scaling rl, 2025. URL `https://pretty-radio-b75.notion.site/DeepSWE-Training-a-Fully-Open-sourced-State-of-the-Art-Coding-Agent-by-Scaling-RL-` Notion Blog.

Yingwei Ma, Rongyu Cao, Yongchang Cao, Yue Zhang, Jue Chen, Yibo Liu, Yuchen Liu, Binhua Li, Fei Huang, and Yongbin Li. Lingma swe-gpt: An open development-process-centric language model for automated software improvement. *arXiv preprint arXiv:2411.00622*, 2024.

Mistral AI. Introducing: Devstral 2 and mistral vibe cli, December 2025. URL `https://mistral.ai/news/devstral-2-vibe-cli`. Accessed: 2026-01-28.

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 *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=Cq1BNvHx74`.

Minh VT Pham, Huy N Phan, Hoang N Phan, Cuong Le Chi, Tien N Nguyen, and Nghi DQ Bui. Swe-synth: Synthesizing verifiable bug-fix data to enable large language models in resolving real-world bugs. *arXiv preprint arXiv:2504.14757*, 2025.

Sijun Tan, Michael Luo, Colin Cai, Tarun Venkat, Kyle Montgomery, Aaron Hao, Tianhao Wu, Arnav Balyan, Manan Roongta, Chenguang Wang, Li Erran Li, Raluca Ada Popa, and Ion Stoica. rllm: A framework for post-training language agents, 2025. URL `https://pretty-radio-b75.notion.site/rLLM-A-Framework-for-Post-Training-Language-Agents-21b81902c146819db63cd98a54ba5f3` Notion Blog.

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.

Konstantinos Vergopoulos, Mark Niklas Mueller, and Martin Vechev. Automated benchmark generation for repository-level coding tasks. In *Forty-second International Conference on Machine Learning*, 2025. URL `https://openreview.net/forum?id=qnE2m3pIAb`.

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*, 2025a.

Lilin Wang, Lucas Ramalho, Alan Celestino, Phuc Anthony Pham, Yu Liu, Umang Kumar Sinha, Andres Portillo, Onassis Osunwa, and Gabriel Maduekwe. Swe-bench++: A framework for the scalable generation of software engineering benchmarks from open-source repositories. *arXiv preprint arXiv:2512.17419*, 2025b.

Ruiyi Wang and Prithviraj Ammanabrolu. A practitioner's guide to multi-turn agentic reinforcement learning. *arXiv preprint arXiv:2510.01132*, 2025.

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

Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, LINGMING ZHANG, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida Wang. SWE-RL: Advancing LLM reasoning via reinforcement learning on open software evolution. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025. URL `https://openreview.net/forum?id=ULb1O61XZ0`.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

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. *arXiv preprint arXiv:2501.05040*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025a.

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. URL `https://openreview.net/forum?id=mXpq6ut8J3`.

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 *The Thirty-ninth Annual Conference on Neural Information Processing*

Systems Datasets and Benchmarks Track, 2025b. URL `https://openreview.net/forum?id=63iVrXc8cC`.

Jiahao Yu, Zelei Cheng, Xian Wu, and Xinyu Xing. Building coding agents via entropy-enhanced multi-turn preference optimization. *arXiv preprint arXiv:2509.12434*, 2025.

Xingdi Yuan, Morgane M Moss, Charbel El Feghali, Chinmay Singh, Darya Moldavskaya, Drew MacPhee, Lucas Caccia, Matheus Pereira, Minseon Kim, Alessandro Sordoni, and Marc-Alexandre Côté. debug-gym: A text-based environment for interactive debugging. *arXiv preprint arXiv:2503.21557*, 2025.

Kechi Zhang, Huangzhao Zhang, Ge Li, Jinliang You, Jia Li, Yunfei Zhao, and Zhi Jin. Sealign: Alignment training for software engineering agent. *arXiv preprint arXiv:2503.18455*, 2025a.

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*, 2025b.

Wanru Zhao, Lucas Caccia, Zhengyan Shi, Minseon Kim, Xingdi Yuan, Weijia Xu, Marc-Alexandre Côté, and Alessandro Sordoni. Learning to solve complex problems via dataset decomposition. In *2nd AI for Math Workshop@ ICML 2025*, 2025.

Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. LlamaFactory: Unified efficient fine-tuning of 100+ language models. In Yixin Cao, Yang Feng, and Deyi Xiong (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 400–410, Bangkok, Thailand, August 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.acl-demos.38. URL `https://aclanthology.org/2024.acl-demos.38/`.

Yiqi Zhu, Apurva Gandhi, and Graham Neubig. Training versatile coding agents in synthetic environments. *arXiv preprint arXiv:2512.12216*, 2025.

Albert Örwall. Moatless Tools, June 2024. URL `https://github.com/aorwall/moatless-tools`.

A Agentic Synthetic Bug Generation

Purposeful Bug Introduction

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python code repository in the directory {{
    working_dir}}.
Your job is to to introduce subtle runtime bugs that cannot be
    reliably detected through code reading alone and require
    debugging tools to diagnose.
The bug you introduce must cause an existing test to fail but
    should require runtime debugging tools (like pdb, breakpoints,
    or state inspection) to diagnose.
It should NOT be detectable through careful code reading or looking
     at the stacktrace of the failing test alone.
Focus on runtime state issues, reference sharing, timing
    dependencies, or complex execution flows that only become
    apparent during execution.

To this end, some kinds of bugs you might introduce include:
- Create data flow bugs through deep object mutation: Modify nested
     data structures (like dictionaries within lists within objects)
     where the mutation path is long and the effect appears far from
     the cause.
- Implement context-dependent behavior with global state pollution:
     Use global variables or class-level state that gets modified as
     a side effect, causing functions to behave differently
    depending on previous execution history.
- Implement recursive functions with shared mutable state: Use
    mutable default arguments or class-level variables in recursive
    functions that accumulate state across different call trees,
    causing interference between separate recursive operations.
- Create shared reference issues with mutable objects: Use the same
     mutable object reference across multiple operations without
    proper copying, causing modifications in one context to
    unexpectedly affect another (e.g., sharing lists or dictionaries
     between instances).
- Introduce accidental state mutations in nested calls: Modify
    object state unexpectedly deep within a chain of method calls,
    where the mutation appears unrelated to the method's stated
    purpose (e.g., a validation method that accidentally modifies
    the object being validated.

Tips for introducing the bug:
- It should not cause compilation errors.
- It should not be a syntax error.
- It should not modify the documentation significantly.
- It should cause a pre-exisiting test to fail. But the bug should
    not be easy to diagnose just by looking at the stacktrace of the
     failing test.
- The root cause should be separated from the symptom manifestation
     - where the bug occurs should be different from where the error
     appears.
- The bug maybe a result of edits to multiple function/files which
    interact in complex ways.
- The bug should require runtime inspection such as stepping
    through execution with a debugger to trace the actual cause - it
     cannot be reliably detected through static code analysis alone.
- For functions with complex state or multiple objects, introduce
    bugs that span multiple method calls or object interactions.
```

```
- Focus on bugs that involve shared state, reference aliasing, or
    side effects that are not immediately obvious but is only
    visible during execution.
- The bug should require tools like pdb, debugger breakpoints, or
    runtime state inspection to diagnose effectively.
- Please DO NOT INCLUDE COMMENTS IN THE CODE indicating the bug
    location or the bug itself.

Follow these steps to introduce the bug:
1. As a first step, it might be a good idea to go over the general
    structure of the repository.
2. Decide where and what kind of bug you want to introduce.
3. Plan out how you might need to make changes to introduce this
    bug.
4. Make the changes by editing the relevant parts of the codebase.
5. Make sure that after editing the code to introduce the bug, at
    least one pre-existing test fails.
6. Make sure that the bug you have introduced cannot be deteced by
    looking at the code or the stacktrace alone, and it need the use
     of debugging tools to diagnose.
7. Do not include any comments in the code or point out the bug in
    any way.
Your thinking should be thorough and so it's fine if it's very long
    .
```

Feature Addition

```
<uploaded_files>
{{working_dir}}
</uploaded_files>
I've uploaded a python code repository in the directory {{
    working_dir}}.

Your task is to implement a new feature in this codebase.
First go through the codebase and identify a suitable new feature
    to add.
Come up with a plan to implement it and then make the necessary
    changes to the codebase.
You can use the tools provided to edit files, run tests, and submit
     your changes.
The feature you introduce should not break any existing
    functionality.
Make sure the edit you make is complex - you should introduce at
    least two related changes in the codebase in different files.
```

B FEATURE ADD EXAMPLE BUGS

Type A - API/signature mismatch or backward-compatibility break

```
ind_available_providers() returns extra provider that breaks
    expected module list

### Describe the bug

The `find_available_providers()` function is returning an
    unexpected provider module `faker.providers.technology` that is
```

~~~
    not part of the expected provider list. This causes issues when
    comparing the actual providers against the expected set.

### How to Reproduce

Run the following code to see the issue:

```python
from faker.utils import find_available_providers
from importlib import import_module
from faker import META_PROVIDERS_MODULES

modules = [import_module(path) for path in META_PROVIDERS_MODULES]
providers = find_available_providers(modules)
expected_providers = [’faker.providers.address’, ’faker.providers.
    automotive’, ’faker.providers.bank’, ’faker.providers.barcode’,
    ’faker.providers.color’, ’faker.providers.company’, ’faker.
    providers.credit_card’, ’faker.providers.currency’, ’faker.
    providers.date_time’, ’faker.providers.emoji’, ’faker.providers.
    file’, ’faker.providers.geo’, ’faker.providers.internet’, ’faker
    .providers.isbn’, ’faker.providers.job’, ’faker.providers.lorem
    ’, ’faker.providers.misc’, ’faker.providers.passport’, ’faker.
    providers.person’, ’faker.providers.phone_number’, ’faker.
    providers.profile’, ’faker.providers.python’, ’faker.providers.
    sbn’, ’faker.providers.ssn’, ’faker.providers.user_agent’]

print("Found providers:", providers)
print("Expected providers:", expected_providers)
print("Match:", providers == expected_providers)
```

**Expected output:**
```
Found providers: [’faker.providers.address’, ’faker.providers.
    automotive’, ’faker.providers.bank’, ’faker.providers.barcode’,
    ’faker.providers.color’, ’faker.providers.company’, ’faker.
    providers.credit_card’, ’faker.providers.currency’, ’faker.
    providers.date_time’, ’faker.providers.emoji’, ’faker.providers.
    file’, ’faker.providers.geo’, ’faker.providers.internet’, ’faker
    .providers.isbn’, ’faker.providers.job’, ’faker.providers.lorem
    ’, ’faker.providers.misc’, ’faker.providers.passport’, ’faker.
    providers.person’, ’faker.providers.phone_number’, ’faker.
    providers.profile’, ’faker.providers.python’, ’faker.providers.
    sbn’, ’faker.providers.ssn’, ’faker.providers.user_agent’]
Expected providers: [’faker.providers.address’, ’faker.providers.
    automotive’, ’faker.providers.bank’, ’faker.providers.barcode’,
    ’faker.providers.color’, ’faker.providers.company’, ’faker.
    providers.credit_card’, ’faker.providers.currency’, ’faker.
    providers.date_time’, ’faker.providers.emoji’, ’faker.providers.
    file’, ’faker.providers.geo’, ’faker.providers.internet’, ’faker
    .providers.isbn’, ’faker.providers.job’, ’faker.providers.lorem
    ’, ’faker.providers.misc’, ’faker.providers.passport’, ’faker.
    providers.person’, ’faker.providers.phone_number’, ’faker.
    providers.profile’, ’faker.providers.python’, ’faker.providers.
    sbn’, ’faker.providers.ssn’, ’faker.providers.user_agent’]
Match: True
```

**Actual output:**
```
Found providers: [’faker.providers.address’, ’faker.providers.
    automotive’, ’faker.providers.bank’, ’faker.providers.barcode’,
    ’faker.providers.color’, ’faker.providers.company’, ’faker.
~~~

```
    providers.credit_card', 'faker.providers.currency', 'faker.
    providers.date_time', 'faker.providers.emoji', 'faker.providers.
    file', 'faker.providers.geo', 'faker.providers.internet', 'faker
    .providers.isbn', 'faker.providers.job', 'faker.providers.lorem
    ', 'faker.providers.misc', 'faker.providers.passport', 'faker.
    providers.person', 'faker.providers.phone_number', 'faker.
    providers.profile', 'faker.providers.python', 'faker.providers.
    sbn', 'faker.providers.ssn', 'faker.providers.technology', '
    faker.providers.user_agent']
Expected providers: ['faker.providers.address', 'faker.providers.
    automotive', 'faker.providers.bank', 'faker.providers.barcode',
    'faker.providers.color', 'faker.providers.company', 'faker.
    providers.credit_card', 'faker.providers.currency', 'faker.
    providers.date_time', 'faker.providers.emoji', 'faker.providers.
    file', 'faker.providers.geo', 'faker.providers.internet', 'faker
    .providers.isbn', 'faker.providers.job', 'faker.providers.lorem
    ', 'faker.providers.misc', 'faker.providers.passport', 'faker.
    providers.person', 'faker.providers.phone_number', 'faker.
    providers.profile', 'faker.providers.python', 'faker.providers.
    sbn', 'faker.providers.ssn', 'faker.providers.user_agent']
Match: False
```

```
### Expected behavior

The `find_available_providers()` function should return only the
    providers that are expected to be in the baseline provider set,
    without including any additional providers like `faker.providers
    .technology` that appear to have been added but aren't part of
    the original expected list.

### Your project

Faker library

### OS

Linux

### Python version

3.10.18

### Additional context

The extra `faker.providers.technology` provider is appearing in the
     returned list at index 24, shifting the expected `faker.
    providers.user_agent` to the end.
```

Type B - Logic/conditional bug

```
`np.row_stack` fails with mixed array shapes in axis=0 mode

Description

When using `np.row_stack` with arrays that have different shapes
    along non-concatenation axes, the operation fails unexpectedly.
    This seems to be a regression as the behavior should match NumPy
    's standard row_stack functionality.
```

~~~
Reproduction:
```python
import autograd.numpy as np

# This should work but fails
arr1 = np.random.random((2, 3))
arr2 = np.random.random((2, 4))
arr3 = np.random.random((1, 4))

result = np.row_stack([arr1, (arr2, arr3)])
~~~

The expected behavior is that `row_stack` should concatenate arrays
     along axis 0, similar to `vstack`. When passed a list
    containing both individual arrays and tuples of arrays, it
    should handle the concatenation properly.

This appears to affect gradient computation as well when used in
    differentiable contexts.
```

**Type C - Input Validation, boundary or sentinel handling error**

```
    Option -a doesn't return expected exit code when invalid
        arguments are provided

Description

The command line option `-a` (which I assume is for setting
    additional arguments or attributes) isn't behaving correctly
    when invalid arguments are passed to it. Instead of returning
    exit code 2 as expected for invalid options, it's returning exit
     code 0.

This seems to be a regression in the command line argument handling
    . When you run the command with `-a arg`, it should fail with
    exit code 2 to indicate invalid usage, but currently it's
    succeeding (exit code 0).

How to reproduce:
```bash
# This should fail with exit code 2 but returns 0 instead
python -m pygments -a arg
echo $?  # prints 0 but should print 2
```

Expected behavior: The command should exit with code 2 when `-a` is
     provided with invalid arguments
Actual behavior: The command exits with code 0

This affects any scripts or CI systems that rely on proper exit
    codes to detect invalid command line usage.
```

**Type D - Incorrect Argument Forwarding**

```
Custom validator repr() shows incorrect class name when created
    with validators.create()

Description
```
```

~~~
When creating a custom validator using `validators.create()`, the `
    repr()` method shows an incorrect class name. Instead of showing
     the actual class name, it displays a formatted version based on
     the version string.

For example:
```python
Validator = validators.create(meta_schema={'$id': 'something'},
    version='my version')
validator = Validator({})
print(repr(validator))
# Shows: MyVersionValidator(schema={}, format_checker=None)
# Expected: <actual class name>Validator(schema={}, format_checker=
    None)
~~~

The repr output uses "MyVersionValidator" instead of the proper
    class name, which makes debugging and introspection more
    difficult when working with custom validators.
```

Type E - Missing import/symbol/attribute error example

```
    ## Pydantic examples in docstrings failing with import errors

Hey folks, I'm running into some issues with the docstring examples
     in pydantic. It looks like there are some import problems
    happening when the examples are being executed.

### Describe the bug

When running docstring examples, some of them are failing during
    execution. The examples seem to be having trouble with imports
    or module resolution. This is affecting the documentation
    validation process.

### How to Reproduce

I created a simple script to reproduce the issue:

```python
import pydantic
from pydantic import BaseModel
from typing import TypeVar, Generic

# Try to run some basic pydantic operations that might be in
    docstrings
T = TypeVar('T')

class MyModel(BaseModel, Generic[T]):
    value: T

# This should work fine normally
model = MyModel[str](value="test")
print(f"Created model: {model}")
```

When this gets executed in the context of docstring evaluation, it
    seems to run into problems.
```
```

```
### Expected behavior

All docstring examples should execute successfully without import
    errors or module resolution issues. The examples are supposed to
     demonstrate proper pydantic usage and should run cleanly.

### Environment

- Python version: 3.10.18
- Pydantic version: Latest from main branch

### Additional context

This seems to be related to how the docstring examples are being
    evaluated and potentially how modules are being imported during
    the evaluation process. The issue appears to affect multiple
    examples across different parts of the codebase.

The problem might be related to the dynamic import system or how
    the evaluation environment is set up for running the docstring
    examples.
```

Type F - State consistency/bookkeeping/caching bug

~~~
**Describe the bug**
Memory leak when checking Union types - objects created in the same
     scope as `check_type()` calls are not being properly garbage
    collected, causing reference leaks.

**To Reproduce**
Create a simple test case with an object that should be garbage
    collected after going out of scope:

```python
from typeguard import check_type
from typing import Union

class TestObject:
    def __del__(self):
        print("Object deleted")

def test_leak():
    obj = TestObject()
    check_type(b'test', Union[str, bytes])
    # Object should be deleted here when function exits

test_leak()
# Expected: "Object deleted" should be printed
# Actual: Nothing is printed, indicating the object wasn't deleted
~~~

Also reproducible with Python 3.10+ union syntax:
```python
def test_leak_new_syntax():
    obj = TestObject()
    check_type(b'test', str | bytes)
    # Object should be deleted here

test_leak_new_syntax()
```

```
```

Expected behavior
Objects should be properly garbage collected when they go out of
    scope, even when `check_type()` is called in the same scope. No
    memory references should be retained by the type checking
    machinery.

Environment info
- Python version: 3.10+ (affects both typing.Union and new union
    syntax)
- typeguard version: latest

Additional context
This appears to affect both the legacy `typing.Union` syntax and
    the newer `str | bytes` union syntax introduced in Python 3.10.
    The issue suggests that the type checking code may be holding
    onto references that prevent proper cleanup of local objects.
```

**Type G - Copy Semantics, mutability aliasing, or in-place mutation of inputs**

```
Contrast improvements break test with `fail_if_improved`
    assertion

I ran into an issue where the contrast test is failing with the
    message "congrats, you improved a contrast! please run ./scripts
    /update_contrasts.py". This happens when the contrast values for
     pygments styles have been improved but the test baseline hasn't
     been updated.

How to Reproduce

The issue occurs when running the contrast tests and some style has
     improved contrast ratios compared to the stored baseline values
    . The test will fail with an assertion error indicating that
    contrasts have improved.

Expected behavior

The test should either automatically update the baseline values
    when improvements are detected, or there should be a clearer way
     to handle contrast improvements without requiring manual script
     execution.

Additional context

The test uses a `fail_if_improved` parameter that's set to `True`
    by default, which causes the test to fail when contrast values
    are better than the stored baseline. This seems counterintuitive
     - improvements in contrast should typically be welcomed rather
    than causing test failures.

The error message suggests running `./scripts/update_contrasts.py`
    but this creates friction in the development workflow when
    contrast improvements happen naturally through code changes.
```
```

Type H - Protocol/spec conformance bug

```
## Bug report

The `tldextract` function and `TLDExtract.extract_str`/`TLDExtract.
    extract_urllib` methods are failing doctest validation. This
    appears to be related to how the doctests are being processed or
     executed.

When running the full test suite, three doctest failures occur:

```
FAILED tldextract/tldextract.py::tldextract.tldextract
FAILED tldextract/tldextract.py::tldextract.tldextract.TLDExtract.
    extract_str
FAILED tldextract/tldextract.py::tldextract.tldextract.TLDExtract.
    extract_urllib
```

The doctests in the main `tldextract` function and the `TLDExtract`
     class methods are not passing validation, while all other
    regular unit tests continue to pass successfully.

This suggests there may be an issue with the expected output
    formatting in the docstrings or how the doctest runner is
    interpreting the examples. The functionality itself seems to
    work correctly based on the passing unit tests, but the embedded
     documentation examples are failing validation.
```

Type I - Resource Mishandling Issue

~~~
When calling the `navigate()` method on a `URL` object with `None`
    as the path parameter, it doesn't behave as expected in certain
    scenarios. This seems to affect URL path resolution when dealing
     with base URLs that have trailing paths.

Here's a minimal reproduction:

```python
from boltons.urlutils import URL

# This works as expected
url = URL('https://host/a/')
result = url.navigate('b')
print(f"Expected: https://host/a/b, Got: {result.to_text()}")

# This doesn't work correctly
url = URL('https://host/a')
result = url.navigate(None).navigate('b')
print(f"Expected: https://host/b, Got: {result.to_text()}")

url = URL('https://host/a/')
result = url.navigate(None).navigate('b')
print(f"Expected: https://host/a/b, Got: {result.to_text()}")
```

Expected behavior:
...
~~~

```
The issue appears to be in how `navigate()` handles `None` paths
    when resolving relative URLs. The method should properly handle
    the case where `None` is passed as a path parameter and maintain
     correct URL resolution behavior for subsequent chained `
    navigate()` calls.

This affects URL manipulation when programmatically building URLs
    where the path might be conditionally `None`.
```

C Bug Categorisation

We use a hierarchical summarisation strategy to come up with bug types to categorise bugs. Bugs from all datasets are pooled together and an LLM is used to come up with summaries of individual bugs along with potential bug types. These summaries are grouped together and further summarised. We continue this process and obtain the following ten bug categories -

Bug Category Descriptions

```
A: API/signature mismatch or backward-compatibility break
  - Description: Public interfaces change or fail to accept/forward
       expected parameters; options no longer propagated; removed/
      renamed methods.
  - Signals: TypeError for unexpected/unknown keyword, missing
      method attribute, inability to customize behavior that used to
       work.
  - Common fixes: Align signatures across layers, add/propagate
      parameters, restore deprecated shims or document breaking
      changes.

B: Logic/conditional bug
  - Description: Incorrect branching, inverted predicates, off-by-
      one comparisons, or misplaced conditions that alter behavior.
  - Signals: Wrong results for specific ranges/cases; behavior
      flips when a flag toggles; regression tied to a refactor of if
      /else logic.
  - Common fixes: Correct predicates/ordering; add minimal repro
      tests around boundary values and both branches.

C: Input validation, boundary, or sentinel handling error
  - Description: Valid inputs rejected or invalid accepted; special
       values (NaN/None/NA/masked) mishandled due to comparison/
      identity semantics.
  - Signals: Edge cases fail while common cases pass; inconsistent
      behavior with empty inputs or special sentinels.
  - Common fixes: Validate before use; use library-appropriate
      checks for sentinels; add edge/empty-case tests.

D: Incorrect argument forwarding, constructor, or inheritance
    contract break
  - Description: Subclasses pass wrong args to super, fail to call
      base initializer, or expose mismatched signatures.
  - Signals: TypeError/AttributeError during object creation;
      missing base attributes; framework hooks not invoked.
  - Common fixes: Align constructor signatures; call super()
      correctly; set attributes after base init; stop forwarding
      unsupported args.

E: Missing import/symbol/attribute error
```

```
  - Description: Required names removed or not imported after
      refactor; attributes expected by callers no longer present.
  - Signals: NameError/AttributeError at runtime; module-level
      failures on import.
  - Common fixes: Restore or re-export symbols; update imports; add
       import-time tests.

F: State consistency/bookkeeping/caching bug
  - Description: Shared or stale state corrupts behavior across
      calls/instances; counters/heaps not updated; cache keys too
      coarse.
  - Signals: Nondeterministic results; memory growth; behavior
      depends on call order; leaked/stale entries.
  - Common fixes: Use per-call/per-instance state; fix increment/
      decrement paths; design proper cache keys; add isolation/
      concurrency tests.

G: Copy semantics, mutability aliasing, or in-place mutation of
    inputs
  - Description: Wrong choice of shallow/deep copy; shared mutable
      defaults; functions mutate caller-provided objects.
  - Signals: Changes in one consumer affect another; unexpected
      side effects; duplicated or missing internal state.
  - Common fixes: Avoid mutating inputs; pick correct copy depth;
      use default_factory for mutables; return defensive copies.

H: Protocol/spec conformance bug
  - Description: Behavior violates external specs (HTTP, OAuth,
      data interchange) or expected wire formats.
  - Signals: Clients reject responses; strict parsers fail; tests
      asserting spec rules break (e.g., HTTP HEAD body handling).
  - Common fixes: Implement per spec; adjust emission/validation
      logic; add conformance tests.

I: IO/filesystem/resource handling bug
  - Description: Incorrect handling of paths/streams/resources;
      special-case short-circuits skip real writes; missing
      directory creation.
  - Signals: Truncated output; OSError/FileNotFoundError; behavior
      differs between stdout vs file.
  - Common fixes: Ensure normal write paths execute; create/check
      dirs; close/flush properly; test both special and normal
      streams.

J: Security/sensitive-data leakage due to logic oversight
  - Description: Credentials/headers applied too broadly (e.g., to
      all domains) or without proper scoping/validation.
  - Signals: Tokens sent to unintended endpoints; security reviews
      flag over-permissive defaults.
  - Common fixes: Scope credentials to allowed domains; enforce
      whitelists; secure defaults; add security-focused tests.
```

We use the following prompt to categorise individual bugs into one of these buckets -

Bug Categorisation Prompt

```
Your task is to categorise a provided bug into a set of given bug
    types.

Here are the guidelines on the bug types -
{guide}
```

```
Here is the bug the needs to be categorised -

<problem_description>
{ps}
</problem_description>

<patch>
{patch}
</patch>

Your response should be in xml format:
<reasoning>
Thinking about which categories that the given bug falls into.
</reasoning>
<category>
Alphabet code of category that bug falls into.
</category>
```

D REINFORCEMENT LEARNING

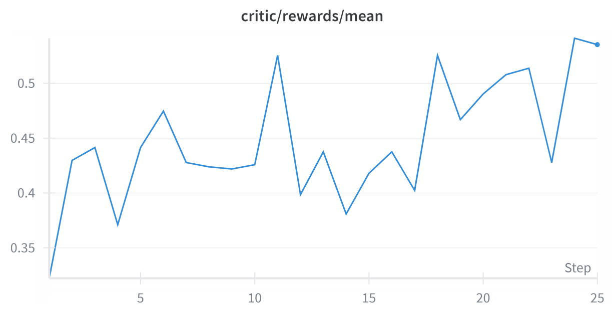

Figure 4: Training reward for reinforcement learning run averaged over batch. We train our reinforcement learning from a base model trained on Qwen3-32B on BASEMIX for 25 steps. Each step consists of 64 problems sampled randomly from FEATADD and 8 rollouts per problem Notably, Claude Sonnet 4 achieves a 41.4% total performance on the FEATADD dataset, but during our training process during the 25th step, the RL process has over 50% training reward. This indicates that the RL model may be better at FEATADD bugs in general over the course of the training process

To train our model with reinforcement learning, we use the rllm framework Tan et al. (2025), an open source paradigm to train reinforcement learning. Previously, this framework was used to train DeepSWE Luo et al. (2025) which achieved an overall performance of 41.0% on SWE-Bench Verified. Our training recipe shows an 11.0% performance improvement over this previous result by bootstrapping from a distilled model with SFT. The main changes in hyperparameter between DeepSWE and our model is the use of 100 max steps and 64k context length. Because the base SFT checkpoint that we use was trained with 100 steps and used a 64k context length to filter to successful trajectories, we believe that this lack of change in paradigm shift is what led the reinforcement learning to train better in this scenario.

Note that when we run with the reinforcement learning we use a max context length of 64k and trim to 32k. We tried one run where we used a max context length of 32k and max steps of 50 and were able to make progress on the training reward but saw a decrease in performance on the evaluation reward on SWE-Bench Verified. Moreover, we tried another run where we finetuned from the base model on a mixture of R2E-GYM and FEATADD bugs but found that the reward was not increasing quickly enough. Contrary to the advice found in Luo et al. (2025), we were able to get a 2.5% improvement over our base SFT model and achieve state-of-the-art results using an SFT+RL paradigm.

Table 5: Hyperparameters for our Reinforcement Learning Run

Hyperparameter	Value
Rollout Temperature	1.0
Max Steps	100
Use KL Loss	False
Train Batch Size	64
Learning Rate	1e-6
PPO Mini Batch Size	8
Max Context Length	64k

Table 6: Comparison along axes of how many lines of code were truly created and how many files were edited. FEATADD bugs involve many more lines and files edited that SWE-SMITH or BUGINSTRUCT. However, about half of the files edited are new files and half of the lines edited are documentation.

	SWE-SMITH	BUGINSTRUCT	FEATADD
Avg. Lines of Code	8.8	14.5	206.5
Avg. Lines of Documentation	4	5.8	233
Avg. Files Edited	1.18	1.50	2.19
Avg. Files Created	0.004	0.09	2.43

E FURTHER DIFF PATCH ANALYSIS

In Table 6, we analyze how many lines of code, lines of documentation and files were edited and/or created in the diff patch generated by SWE-SMITH, BUGINSTRUCT, and FEATADD.

F RESULTS ON HARDER SUBSETS OF SWE-BENCH VERIFIED

In Table 7, we present results of models fine-tuned on various bug data mixes evaluated on harder subsets of SWE-Bench Verified as specified here[1].

While SWE-Bench Verified has been extensively studied, subsets have been identified that drive most of the progress for state-of-the-art models[2]. Namely, the Frontier, Challenging, Hard (judged by experts to require more than one hour to solve each), and Multi-File problems are problems on which Claude Opus 4 achieves 11%, 31%, 42.2% and 10.0% respectively, despite achieving 73.60% on the entire set of SWE-Bench Verified. We find that on all of these harder subsets, training with BASEMIX + FEATADD consistently results in improvements over just BASEMIX. Training with BASEMIX + FEATADD also results in better performance than FEATADD + BUGINSTRUCT in the Challenging and Frontier subsets. Detailed results present in Appendix F. These results indicate that including FEATADD during training helps improve performance across a broad range of hard tasks.

G EXAMPLE OF BUG GENERATION APPROACHES

[1]Subsets from SWE-bench_Verified-discriminative
[2]Subsets from SWE-bench_Verified-discriminative

Table 7: **Results on Harder Subsets of SWE-Bench Verified.** We report the Pass@1 averaged over three seeds. The frontier, challenging, hard, and multi-file subsets contain problems where state-of-the-art closed source models struggle.

Size	Full 500	Challenging 155	Frontier 95	Hard 45	Multi-file 40
Qwen3-32B	25.33	1.29	0.35	5.19	0.83
BaseMix (SFT)	49.87	3.66	1.05	12.59	0.83
BaseMix + R2E-Gym (SFT)	50.73	5.38	1.75	9.63	2.50
BaseMix + SWE-Smith (SFT)	50.57	5.59	2.11	15.56	**2.50**
BaseMix + FeatAdd (SFT)	51.93	6.45	**2.81**	14.07	1.67
BaseMix + FeatAdd (RL)	52.40	5.81	0.35	**15.56**	0.83
AllData (FrogBoss, SFT)	**54.60**	**7.96**	1.05	14.81	1.67

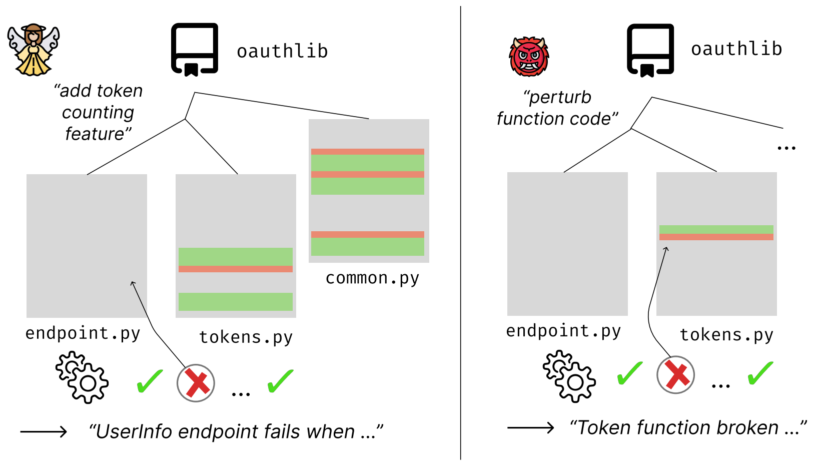
oauthlib
"add token counting feature"
common.py
endpoint.py
tokens.py
→ "UserInfo endpoint fails when ..."
oauthlib
"perturb function code"
...
endpoint.py
tokens.py
→ "Token function broken ..."

Figure 5: **Example of contrasting approaches to adding bugs**. On the left, our FeatAdd approach attempts to implement a token counting feature in the repository. This results in large changes across multiple files as well as a test case failure arising from a seemingly unrelated part of the repository. In contrast on the right, approaches like SWE-Smith and our proposed baseline BugInstruct make local perturbations to the code which cause related tests to fail. We can see that FeatAdd more closely resembles how bugs arise during the development process, where test failures can occur due to complex interactions between changes.