SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution

Yuxiang Wei[1,2], **Olivier Duchenne**[1], **Jade Copet**[1], **Quentin Carbonneaux**[1]
Lingming Zhang[2], **Daniel Fried**[3,4], **Gabriel Synnaeve**[1], **Rishabh Singh**[3], **Sida I. Wang**[1]

[1]FAIR at Meta, [2]University of Illinois Urbana-Champaign
[3]GenAI at Meta, [4]Carnegie Mellon University

The recent DeepSeek-R1 release has demonstrated the immense potential of reinforcement learning (RL) in enhancing the general reasoning capabilities of large language models (LLMs). While DeepSeek-R1 and other follow-up work primarily focus on applying RL to competitive coding and math problems, this paper introduces SWE-RL, the first approach to scale RL-based LLM reasoning for real-world software engineering. Leveraging a lightweight rule-based reward (e.g., the similarity score between ground-truth and LLM-generated solutions), SWE-RL enables LLMs to autonomously recover a developer's reasoning processes and solutions by learning from extensive open-source software evolution data—the record of a software's entire lifecycle, including its code snapshots, code changes, and events such as issues and pull requests. Trained on top of Llama 3, our resulting reasoning model, Llama3-SWE-RL-70B, achieves a **41.0%** solve rate on SWE-bench Verified—a human-verified collection of real-world GitHub issues. To our knowledge, this is the best performance reported for medium-sized (<100B) LLMs to date, even comparable to leading proprietary LLMs like GPT-4o. Surprisingly, despite performing RL solely on software evolution data, Llama3-SWE-RL has even emerged with generalized reasoning skills. For example, it shows improved results on five out-of-domain tasks, namely, function coding, library use, code reasoning, mathematics, and general language understanding, whereas a supervised-finetuning baseline even leads to performance degradation on average. Overall, SWE-RL opens up a new direction to improve the reasoning capabilities of LLMs through reinforcement learning on massive software engineering data.

Date: February 26, 2025
Code: https://github.com/facebookresearch/swe-rl

Meta

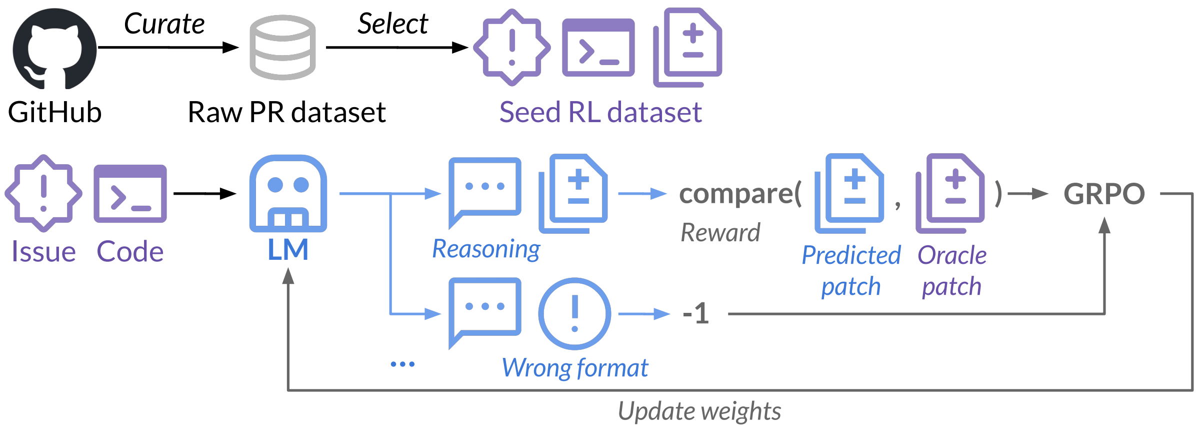


Figure 1 Overview of SWE-RL. We create a seed RL dataset from GitHub PR data, including issue descriptions, code context, and oracle patches. The policy LLM generates code changes through reasoning. For correctly formatted responses, the reward is calculated based on the match between the predicted and the oracle patch; incorrectly formatted responses are assigned a negative reward. GRPO is used for policy optimization.

1 Introduction

The application of large language models (LLMs) to software engineering (SE) tasks has received significant attention in recent years, with researchers exploring their potential to automate various complex SE tasks, such as library-level and complex code generation (Zhuo et al., 2024; Zhao et al., 2024), real-world bug/issue resolution (Xia and Zhang, 2022; Jimenez et al., 2023; Yang et al., 2024c), and software testing (Deng et al., 2023; Jain et al., 2024a). Among these tasks, SWE-bench (Jimenez et al., 2023)—a benchmark for solving real-world software issues—has emerged as a focal point of research efforts, and researchers have proposed various agentic (Yang et al., 2024b; Wang et al., 2024; Gauthier, 2024) or pipeline-based (Xia et al., 2024; Örwall, 2024; Zhang et al., 2024) methods to push LLMs' real-world issue solving capability. However, most current techniques depend on powerful proprietary LLMs like GPT-4o (OpenAI, 2024a) or Claude-3.5-Sonnet (Anthropic, 2024a), where advancements are driven more by enhanced prompting strategies than by improvements in the underlying LLM.

With the release of DeepSeek-R1 (DeepSeek-AI, 2025), reinforcement learning (RL) using rule-based rewards has become a crucial technique for enhancing the reasoning capabilities of LLMs across various downstream tasks, including coding (Zeng et al., 2025) and mathematics (Yeo et al., 2025). However, their effectiveness in SE tasks remains limited (DeepSeek-AI, 2025), and their substantial total parameter size (671B) poses challenges for researchers attempting to train them. For mathematics, the reward is generally defined as whether the answer predicted by the LLM can exactly match the ground truth (DeepSeek-AI, 2025). While in coding, existing RL research typically utilizes execution feedback (Gehring et al., 2025) as the reward signal and is limited to competitive programming tasks (Li et al., 2022; Jain et al., 2024b), where code is self-contained and easily executable. This is challenging to apply to real-world SE tasks due to the execution cost and lack of executable environments (Pan et al., 2024). Meanwhile, previous research (Ma et al., 2024; Pan et al., 2024; Xie et al., 2025) relied on proprietary teacher models and has focused primarily on supervised fine-tuning (SFT), which, as we demonstrate in our paper, is less effective and less generalizable.

To address these limitations, we propose SWE-RL, the first RL method to improve LLMs on SE tasks by directly using rule-based rewards and software evolution data—the record of entire software lifecycle, including all code snapshots, changes, and events like PRs and issues. As shown in Figure 1, we begin by curating a comprehensive dataset of GitHub pull requests (PRs), which is then transformed into the seed dataset for RL. Each data item includes an issue, the corresponding code context, and the oracle patch merged by the PR. During RL, the policy LLM is tasked with solving a given issue through reasoning and producing the code changes. The code changes are then converted into a consistent patch format for reward calculation. If the response is incorrectly formatted, the reward will be -1; otherwise, the reward is a similarity score (between 0 and 1) of the predicted and the oracle patch calculated by Python's `difflib.SequenceMatcher` (Ratcliff and Metzener, 1988). Notably, we provide the complete content of each file in the input prompt, which implicitly teaches the model to reason about the precise fault locations before suggesting repair edits.

Applying SWE-RL to Llama-3.3-70B-Instruct (Dubey et al., 2024), our model Llama3-SWE-RL-70B solves **41.0%** of the issues in SWE-bench Verified (OpenAI, 2024), a human-verified subset of SWE-bench, with Agentless Mini, our pipeline-based scaffold built upon Agentless (Xia et al., 2024), featuring simplifications to match our RL process and enhancements for scaling. This performance is comparable to leading proprietary LLMs like GPT-4o (OpenAI, 2024a) and state-of-the-art among medium-sized LLMs with less than 100B total parameters. Our ablation studies demonstrate that Llama3-SWE-RL-70B significantly outperforms its Llama baseline. Additionally, we developed a competitive supervised fine-tuning (SFT) model from Llama-3.3-70B-Instruct using synthetic data generated in the Magicoder (Wei et al., 2024) style to enhance the chain-of-thought (Wei et al., 2022) process, employing the same seed as SWE-RL. We show that Llama3-SWE-RL-70B, trained with SWE-RL solely for solving issues, not only surpasses the SFT model in SWE-bench but also excels in other out-of-domain (OOD) tasks, including function-level coding (Chen et al., 2021; Liu et al., 2023), practical code generation with library use (Zhuo et al., 2024), code reasoning (Gu et al., 2024), mathematics (Hendrycks et al., 2021c), and general language understanding (Hendrycks et al., 2021b). In these OOD tasks, Llama3-SWE-RL-70B even outperforms Llama-3.3-70B-Instruct, whereas the SFT model results in decreased performance.

In summary, our contributions are as follows:

- We introduce SWE-RL, the first RL approach specifically designed to enhance LLMs for SE tasks using software evolution data (e.g., PRs) and rule-based rewards.
- We develop Llama3-SWE-RL-70B, trained with SWE-RL on Llama-3.3-70B-Instruct. It achieves **41.0%** on SWE-bench Verified, the best performance among medium-sized language models (<100B) and even comparable to leading proprietary models like GPT-4o.
- We show for the first time that applying RL solely to real-world SE tasks, such as issue solving, can already enhance an LLM's general reasoning abilities, enabling it to improve on out-of-domain tasks like math, code generation, and general language understanding.

2 SWE-RL

2.1 Raw pull request data curation

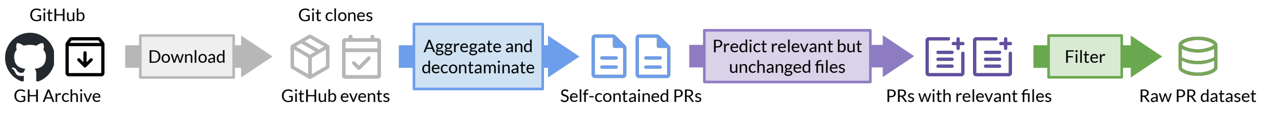


Figure 2 Overview of SWE-RL's raw pull request data curation process. The collected git clones and GitHub events are transformed into self-contained PR instances via decontamination, aggregation, relevant files prediction, and filtering.

Figure 2 provides a high-level overview of our process for curating the raw PR dataset for Llama3-SWE-RL. In the following paragraphs, we detail each step of the curation process. During data processing, we exclude all the repositories used by SWE-bench (Jimenez et al., 2023) to prevent data contamination.

GitHub events and clones. The goal of this stage is to recover all pull request details that human developers can inspect on GitHub. To achieve this, we need two sources of information: (1) all events that occur within a PR and (2) the source code of a repo before the changes introduced by the PR are merged. We derive all GitHub (GitHub, 2025) events from GHArchive (Grigorik, 2025), which contains all activity events data from GitHub. Our collection includes all GitHub events from Jan 1, 2015 to Aug 31, 2024.

To obtain source code, since pull requests often occur at different commit stages of a repository, we opt to use `git clone` to retrieve the entire repository with its commit history, rather than relying on the GitHub API (GitHub, 2022) to download specific code snapshots. Eventually, we successfully cloned and processed 4.6M repositories.

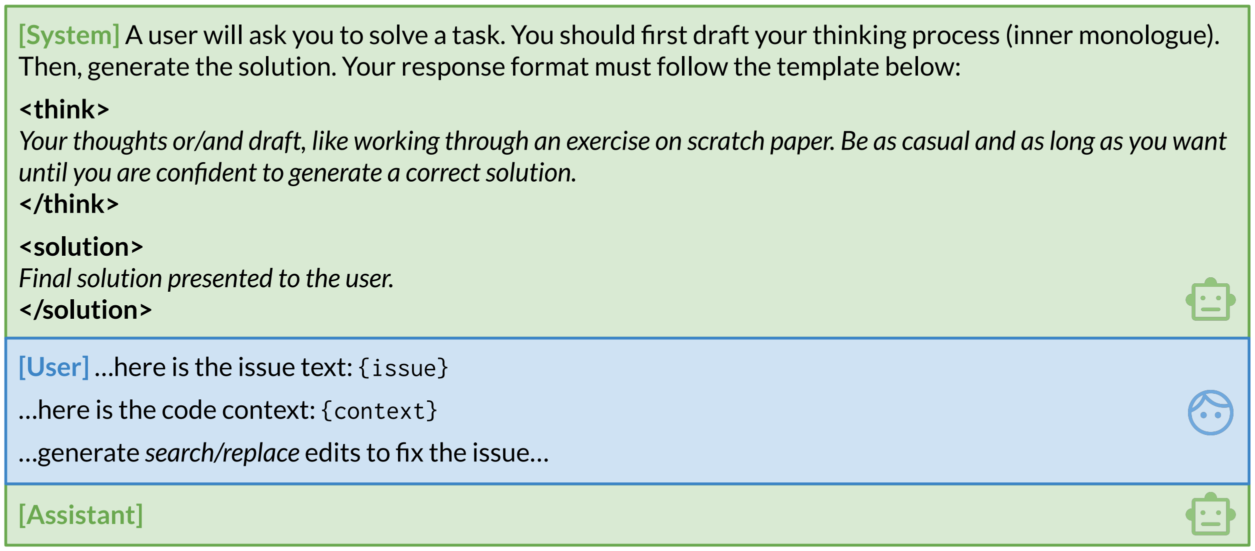
PR data aggregation. These collected events and git clones are disparate entities that require further processing before they can be used for training purposes. At this stage, we focus on each PR individually and aggregate all pertinent information associated with it. This includes mentioned issues, user discussions, review comments, initial code contents, and subsequent commits and code changes.

To start, we keep only merged PRs and gather all related conversational events for each PR, sorting them in chronological order. Next, using the `base_commit` and `head_commit` hashes of a PR, we retrieve the contents of all modified files indicated by its patch at the merge base of the two commits. The reasoning behind this approach is that many PRs aim to merge back into the main branch, which may have undergone changes since the PR was created. By considering the merge base as the actual starting point for a developer working on the PR, we can more accurately understand the context of the changes. We save all intermediate commits and code changes made between the merge base and the head commit. Additionally, we extract the complete patch that represents the cumulative changes from start to finish. Finally, we scan each aggregated PR to identify patterns that resemble issues, and associate the matched issues with the corresponding PR. In the end, we have 24M aggregated PR instances.

Relevant files prediction. Currently, each pull request includes only the code files that have been modified. In our earlier experiments, we noticed that this approach let LLMs learn a bias: the model consistently generated edits for every code file presented and was unable to handle noisy files presented in the context. This issue was also mentioned in the finetuning experiment discussed in the SWE-bench paper (Jimenez et al., 2023). To mitigate this problem, we prompt Llama-3.1-70B-Instruct (Dubey et al., 2024) to generate a list of relevant but unmodified files for each PR, given the PR description and the paths of the changed files, and include the contents of these files in our final dataset.

Data filtering. GitHub PRs can be quite noisy, so we implemented various filtering strategies to eliminate potentially harmful PRs. In designing these filtering rules, our goal is to maximize the recall of high-quality PRs while permitting a certain level of noise. First, we remove the bot-generated PRs whose title, description, or username contains keywords "[bot]", "dependabot", "renovate", "bump", or "automerge". Also, we remove PRs with empty changes or with extremely large number of changes (e.g., in some PRs, the developer uploaded a directory of data files by mistake). Additionally, we implemented a more fine-grained set of filters used in CodeLlama (Rozière et al., 2023) to examine each code change hunk. We then removed any PRs where all code changes were flagged by these filters. For example, this can exclude PRs with only lock file changes or version updates. Finally, this gives us around 11M unique PR instances.

2.2 Reward modeling



[System] A user will ask you to solve a task. You should first draft your thinking process (inner monologue). Then, generate the solution. Your response format must follow the template below:

<think>
Your thoughts or/and draft, like working through an exercise on scratch paper. Be as casual and as long as you want until you are confident to generate a correct solution.
</think>

<solution>
Final solution presented to the user.
</solution>

[User] ...here is the issue text: `{issue}`

...here is the code context: `{context}`

...generate *search/replace* edits to fix the issue...

[Assistant]

Figure 3 Prompt template used to train Llama3-SWE-RL with SWE-RL. Given an issue description and the corresponding code context, the policy LLM needs to generate search/replace edits (Xia et al., 2024) to fix this issue through reasoning. This is the only subtask we incorporate in the RL training. During inference, the LLM can generalize to tasks outside the training domain (e.g, file-level localization and test generation). For conciseness, we exclude certain prompt details, with the complete prompt template available in Appendix D.

To prepare the initial dataset for RL, we extract high-quality PR seeds from the raw dataset we collected. These seeds are selected based on specific heuristics. For example, a PR instance should include at least one linked issue, the issue should describe a bug-fixing request, and the code changes should involve programming files. For each seed, we extract the issue descriptions and code context, including all changed files and some relevant but unchanged files. These are converted to input prompts for the policy LLM. We also take the oracle patch from each seed, which will be used for reward calculation.

We bootstrap the policy LLM with the prompt template shown in Figure 3. Assuming that the LLM has generated a rollout τ given an issue and its code context, the reward function is defined as follows:

$$\mathcal{R}(\tau) = \begin{cases} -1, & \text{if the format is wrong,} \\ \mathit{compare}(\mathsf{patch}_{\mathsf{pred}}, \mathsf{patch}_{\mathsf{gt}}), & \text{otherwise.} \end{cases} \quad (1)$$

Here, $\mathsf{patch}_{\mathsf{pred}}$ denotes the patch corresponding to the code changes generated by the policy LLM if the format of τ is correct, and $\mathsf{patch}_{\mathsf{gt}}$ means the oracle patch for this issue.

In the implementation, we instantiate the *compare* function with Python's `difflib.SequenceMatcher`, which returns a floating point between 0 and 1 indicating the sequence similarity between two sequences, and use Group Relative Policy Optimization (GRPO) (Shao et al., 2024) for policy optimization.

Given a seed RL dataset $\mathcal{D}_{\mathsf{seed}}$ where each item contains (1) issue, the issue description, (2) ctx, the code context required to solve this issue, including both files to repair and relevant files not intended for editing, and (3) $\mathsf{patch}_{\mathsf{gt}}$, which is the oracle patch. The input prompt for each data item is formed by instantiating the prompt template in Figure 3 with the issue description and code context, which we denote as $q = \mathsf{form\text{-}prompt}(\mathsf{issue}, \mathsf{ctx})$. The policy LLM π_θ tries to solve the issue by generating code changes through reasoning, and produces multiple outputs o_i for each input prompt q given the group size G. Then, the policy LLM aims to maximize the following GRPO objective:

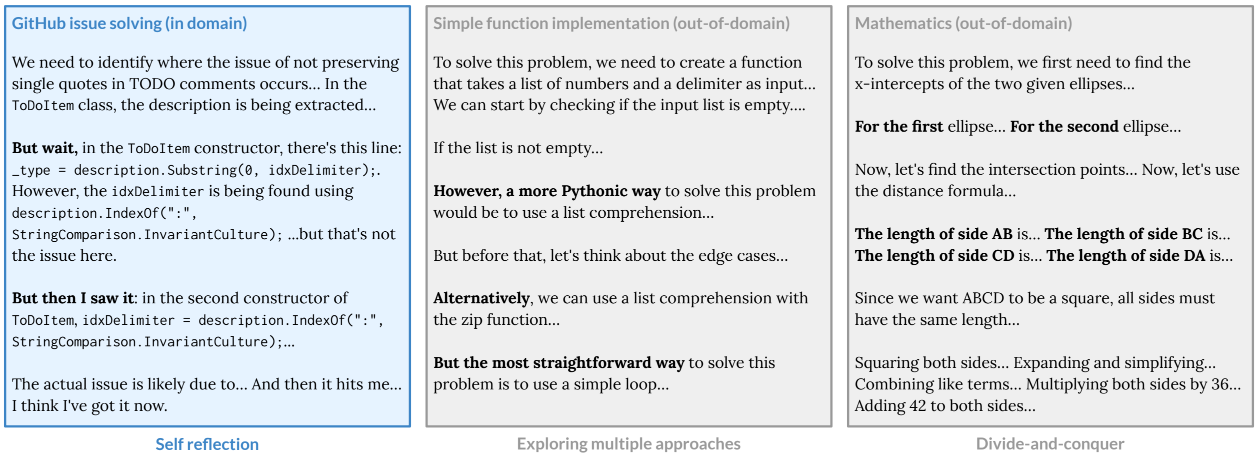
$$\mathcal{J}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\left(\min\left(\frac{\pi_\theta(o_i \mid q)}{\pi_{\theta_{\text{old}}}(o_i \mid q)}A_i, \text{clip}\left(\frac{\pi_\theta(o_i \mid q)}{\pi_{\theta_{\text{old}}}(o_i \mid q)}, 1-\epsilon, 1+\epsilon\right)A_i\right) - \beta D_{\text{KL}}(\pi_\theta \,\|\, \pi_{\text{ref}})\right)\right], \quad (2)$$

$$\text{where } (\mathsf{issue}, \mathsf{ctx}, \mathsf{patch}_{\mathsf{gt}}) \sim \mathcal{D}_{\mathsf{seed}},\ q = \mathsf{form\text{-}prompt}(\mathsf{issue}, \mathsf{ctx}), \text{ and } \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid q).$$

In this equation, ϵ and β are hyperparameters, and $\pi_{\theta_{\text{old}}}$ and π_{ref} are the old and reference policy, respectively. Following GRPO, the advantages A_i are calculated using the normalized rewards within each group. The term D_{KL} denotes the approximated KL-divergence (Schulman, 2020).

Our training approach conditions the model on the complete context of each file, implicitly forcing it to identify detailed fault locations before generating repair edits. This process inherently teaches the model both bug diagnosis and repair generation. However, during downstream evaluation on SWE-bench, our Agentless Mini scaffold requires capabilities beyond generating repair edits, such as file-level fault localization, reproduction test generation, and regression test selection. Remarkably, even without explicit training on these subtasks, Llama3-SWE-RL can generalize to them through the RL process.

2.3 Aha moments and generalized reasoning capabilities


GitHub issue solving (in domain)

We need to identify where the issue of not preserving single quotes in TODO comments occurs... In the `ToDoItem` class, the description is being extracted...

But wait, in the `ToDoItem` constructor, there's this line: `_type = description.Substring(0, idxDelimiter);`. However, the `idxDelimiter` is being found using `description.IndexOf(":", StringComparison.InvariantCulture);` ...but that's not the issue here.

But then I saw it: in the second constructor of `ToDoItem`, `idxDelimiter = description.IndexOf(":", StringComparison.InvariantCulture);`...

The actual issue is likely due to... And then it hits me... I think I've got it now.

Self reflection

Simple function implementation (out-of-domain)

To solve this problem, we need to create a function that takes a list of numbers and a delimiter as input... We can start by checking if the input list is empty....

If the list is not empty...

However, a more Pythonic way to solve this problem would be to use a list comprehension...

But before that, let's think about the edge cases...

Alternatively, we can use a list comprehension with the zip function...

But the most straightforward way to solve this problem is to use a simple loop...

Exploring multiple approaches

Mathematics (out-of-domain)

To solve this problem, we first need to find the x-intercepts of the two given ellipses...

For the first ellipse... **For the second** ellipse...

Now, let's find the intersection points... Now, let's use the distance formula...

The length of side AB is... **The length of side BC** is...
The length of side CD is... **The length of side DA** is...

Since we want ABCD to be a square, all sides must have the same length...

Squaring both sides... Expanding and simplifying...
Combining like terms... Multiplying both sides by 36...
Adding 42 to both sides...

Divide-and-conquer

Figure 4 Reasoning skills emerged from Llama3-SWE-RL-70B following the application of SWE-RL. RL helps the model develop reasoning skills like self-reflection, exploring alternatives, and divide-and-conquer strategies, for both in-domain (e.g., issue-solving) and out-of-domain tasks (e.g., function implementation and mathematics).

"Aha moments" on software engineering. With the application of SWE-RL, we observe "aha moments" (DeepSeek-AI, 2025) where Llama3-SWE-RL exhibits emergent reasoning skills. To our knowledge, this is the first study demonstrating the existence of such aha moments in the realm of real-world software engineering tasks, confirming the findings of DeepSeek-R1 (DeepSeek-AI, 2025), which mainly focuses on competition coding and math. As shown in Figure 4, through RL, Llama3-SWE-RL-70B can allocate more thinking time to reflect on its initial assumptions during the issue-solving process. This behavior emerges naturally from the model's interaction with RL, rather than being explicitly programmed.

General reasoning capabilities. Surprisingly, we have identified additional aha moments where Llama3-SWE-RL acquires general reasoning abilities that are transferrable to various out-of-domain tasks, such as function-level code generation and mathematics, although RL is applied exclusively to software issue solving. Figure 4 demonstrates that Llama3-SWE-RL is capable of reasoning through self-reflection, exploring alternative approaches, and solving complex problems by breaking them down into smaller subtasks. In §3.5, we demonstrate that Llama3-SWE-RL improves over even more out-of-domain tasks, including library use, code reasoning, and general language understanding.

3 Evaluation

3.1 Experimental setup

Training configs. Llama3-SWE-RL-70B is trained on top of Llama-3.3-70B-Instruct (Dubey et al., 2024) using SWE-RL for 1,600 steps with a 16k context window. We use a global batch size of 512, sampling 16 rollouts from each of the 32 problems in every batch. For every global step, a single optimization step is performed.

Scaffolding. We have developed Agentless Mini on top of Agentless (Xia et al., 2024) as the underlying scaffold. Different from Agentless's multi-step localization, Agentless Mini focuses solely on file-level localization, delegating detailed reasoning to the repair step by providing the entire file contents in the input. This enables Llama3-SWE-RL to do more reasoning during SWE-RL and simplifies the RL process to focus on only one issue-solving task. Despite this simplification, Llama3-SWE-RL can still seamlessly complete other pipeline steps, benefiting from out-of-domain generalization through RL. Furthermore, while Agentless only employs one reproduction test per issue for reranking, Agentless Mini can use multiple reproduction tests, which has proven effective in our scaling analysis (§3.4). More details about Agentless Mini can be found in Appendix A.

Evaluation setup. We conduct evaluation on SWE-bench Verified (OpenAI, 2024), a subset of SWE-bench with 500 human-verified problems that can more reliably evaluate AI models' capability in solving real-world software issues. In the main evaluation (§3.2), we generate 500 patches for each problem using a 1.0 temperature, and use the top 30 reproduction tests for execution and reranking. Ultimately, only the highest-ranked patch for each issue will be submitted for SWE-bench evaluation to calculate the pass@1 score.

SFT baseline. To understand the advantages of SWE-RL, we also trained an SFT baseline, named Llama3-SWE-SFT-70B, for experiments in §3.3, §3.4, and §3.5. It is trained on top of Llama-3.3-70B-Instruct (Dubey et al., 2024) using a mixture of synthetic code editing data, Llama 3 (Dubey et al., 2024) coding SFT data, and Llama 3 general SFT data. The synthetic data is generated using an approach inspired by Magicoder (Wei et al., 2024), where high-quality PR data serves as the seeds for creating chain-of-thoughts and subsequent editing, as well as serving as the oracle for filtering. Llama-3.3-70B-Instruct is employed for this generation process. The seed PR dataset is identical to the one utilized for RL. In contrast to our RL model, which only requires a seed PR dataset to trigger the RL loop, the SFT baseline needs synthetic data generation for chain-of-thoughts and additional data mix to ensure the dataset diversity and model generalizability. More details are explained in Appendix B.

3.2 Main results

Table 1 presents the pass@1 results on SWE-bench Verified for models that utilize open-source scaffolds, with models categorized by their size. From the table, we observe that Llama3-SWE-RL-70B achieves state-of-the-art results among small and medium-sized language models (<100B) by resolving 41.0% of the issues. Additionally, all other open-source baselines we compare, such as Lingma-SWE-GPT (Ma et al., 2024), SWE-Gym (Pan et al., 2024), and SWE-Fixer (Xie et al., 2025), include distilled outputs from GPT-4o or Claude-3.5-Sonnet in the training data. In contrast, Llama3-SWE-RL is trained solely with publicly available data through our reinforcement learning technique SWE-RL, without relying on any proprietary LLMs in the pipeline. Llama3-SWE-RL also sets a new record for Llama-based methods on SWE-bench.

Model	Scaffold	SWE-bench Verified	Reference
Model closed-source or size $\gg$ 100B			
GPT-4o	SWE-agent	23.2	Yang et al. (2024b)
Claude-3.5-Sonnet	SWE-agent	33.6	Yang et al. (2024b)
GPT-4o	Agentless	38.8	Xia et al. (2024)
o1-preview	Agentless	41.3	OpenAI (2024b)
DeepSeek-V3[1]	Agentless	42.0	DeepSeek-AI (2024)
Claude-3.5-Sonnet	AutoCodeRover-v2.0	46.2	Zhang et al. (2024)
Claude-3.5-Sonnet	Tools	49.0	Anthropic (2024b)
DeepSeek-R1[1]	Agentless	49.2	DeepSeek-AI (2025)
Claude-3.5-Sonnet	Agentless	50.8	Xia et al. (2024)
Claude-3.5-Sonnet	OpenHands	53.0	Wang et al. (2024)
Model size $\leq$ 100B			
SWE-Llama-13B	RAG	1.2	Jimenez et al. (2023)
SWE-Llama-7B	RAG	1.4	Jimenez et al. (2023)
Lingma-SWE-GPT-7B	SWE-SynInfer	18.2	Ma et al. (2024)
Lingma-SWE-GPT-72B	SWE-SynInfer	28.8	Ma et al. (2024)
SWE-Fixer-72B	SWE-Fixer	30.2	Xie et al. (2025)
Llama3-Midtrain-8B (beta)[2]	**Agentless Mini**	**31.0**	**Appendix C**
SWE-Gym-32B	OpenHands	32.0	Pan et al. (2024)
Llama3-SWE-RL-70B	**Agentless Mini**	**41.0**	**This paper**

[1]Open-source Mixture-of-Experts model with 671B total and 37B active parameters
[2]A beta version developed via extensive midtraining on raw PR data. See Appendix C.

Table 1 Main results on SWE-bench Verified. We include representative methods with open-source scaffolds. The scores are either collected from the SWE-bench Leaderboard (Jimenez et al., 2024) or from the corresponding reference.

Model	Setting	Correct format	Repair performance (oracle)
Llama-3.3-70B-Instruct	Greedy decoding	12.2%	5.4
Llama-3.3-70B-Instruct	Majority voting	44.6%	16.6
Llama3-SWE-SFT-70B	Greedy decoding	**96.2%**	29.6
Llama3-SWE-RL-70B	**Greedy decoding**	95.6%	**34.8**

Table 2 Baseline comparison on SWE-bench Verified. In this experiment, we compare the repair-only performance of baseline LLMs by providing oracle localized files in the input context, without doing test generation and execution. We use greedy decoding by default, but for Llama-3.3-70B-Instruct, we include a 20-sample majority voting result at a temperature of 0.6 to improve formatting accuracy.

3.3 Baseline comparison

To understand how much SWE-RL improves LLMs in solving sofware issues, we compare Llama3-SWE-RL with the corresponding Llama-3 and SFT baseline in Table 2, using Agentless Mini as the underlying scaffold. In this experiment, we also evaluate on SWE-bench Verified but focus on the models' repair ability. To achieve this, we provide oracle files in the context and let the model generate a single repair edit using greedy decoding, without incorporating additional pipeline steps such as localization and test generation. The table reveals that the base Llama-3.3 model struggles to produce correctly formatted code edits, even when using a 20-sample majority voting approach, where outputs with incorrect formats are pre-filtered. With SFT, most code edits generated by the language model are correctly formatted, and the repair performance shows significant improvement. However, Llama3-SWE-RL-70B demonstrates even greater enhancement in repair capabilities, although its format accuracy is slightly lower than that of the SFT version. This indicates that SWE-RL aids the LLM in better reasoning about issue solving and code editing.

3.4 Scaling analysis with more samples

Agentless Mini supports scaling both the number of repair samples and the number of generated reproduction tests. The difference in the number of samples may affect the reranking accuracy. In this section, we evaluate how the final pass@1 performance on SWE-bench Verified scales with the two factors.

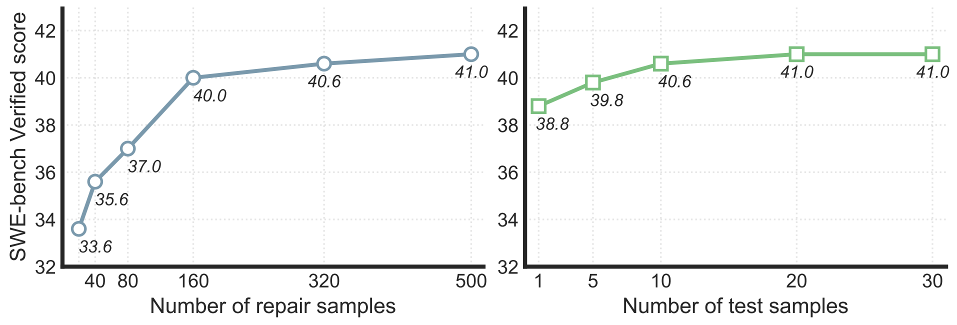


Figure 5 Scaling analysis with more repair samples and more reproduction tests. The figure on the left illustrates the resolve rate on SWE-bench Verified in relation to the number of repair samples, while maintaining a constant 30 test samples. Conversely, the figure on the right depicts the resolve rate as it varies with the number of reproduction test samples, with a fixed count of 500 repair samples.

Figure 5 shows that increasing both the number of repair samples and test samples enhances performance on SWE-bench. Notably, for repair samples, there is a significant score increase from 33.6 to 40.0 when the sample size is expanded from 20 to 160. However, beyond 160 samples, the improvement trend begins to plateau, with scores only rising slightly from 40.0 to 41.0 as the sample size increases to 320 and 500. Although the impact of adding more reproduction test samples is less obvious, there is still a gradual score improvement from 38.8 to 41.0 as the number of test samples increases up to 20. There is no difference between 20 and 30 test samples, suggesting a performance saturation point has been reached.

3.5 Generalizability of RL

Category **Benchmark**	Llama-3.3-70B-Instruct	Llama3-SWE-SFT-70B	**Llama3-SWE-RL-70B**
Function coding			
HumanEval+	76.2	73.2	**79.9**
Library use			
BigCodeBench-Hard (I)	**28.4**	25.7	**28.4**
BigCodeBench-Hard (C)	**29.1**	24.3	**29.1**
Code reasoning			
CRUXEval-I	60.5	68.4	**71.6**
CRUXEval-O	61.9	75.1	**75.5**
Math			
MATH (strict)	63.2	54.0	**73.7**
MATH (lenient)	70.9	71.7	**73.7**
General			
MMLU	86.49	85.26	**86.82**

Table 3 Generalizability of Llama3-SWE-RL-70B beyond SWE-bench. This table compares Llama-3.3-70B-Instruct, the SFT variant, and the RL model on five out-of-domain tasks, highlighting RL improvements and SFT declines. All experiments are done in a consistent setting using zero-shot greedy decoding. We report the macro average over category accuracy for MMLU and pass@1 for the others. In MATH, we use simple-evals's "`Answer: ...`" prompt format (OpenAI, 2024). However, only the RL model consistently follows the format requirements, so we also report MATH (lenient) to relax the constraint to include "`\boxed...`".

Llama3-SWE-RL is only trained with SWE-RL on issue-solving data. This raises a natural question whether such domain-specific training harms the performance on other tasks. To address this, we conduct an experiment in Table 3, evaluating the LLMs on five out-of-domain benchmarks, i.e., HumanEval+ (Chen et al., 2021; Liu et al., 2023) for function-level code generation, BigCodeBench (Zhuo et al., 2024) for practical code generation with library use, CRUXEval (Gu et al., 2024) for code execution reasoning, MATH (Hendrycks et al., 2021c) for mathematical reasoning, and MMLU (Hendrycks et al., 2021b) for general language understanding. We also include the SFT baseline, which is finetuned on the same Llama-3.3-70B-Instruct model using issue-solving data, combined with general coding and dialog data.

From the table, it is evident that Llama3-SWE-RL-70B, trained with RL, outperforms both its base model and the SFT baseline. There are notable improvements in CRUXEval and MATH, where significant reasoning efforts are required to arrive at the final answer. Through SWE-RL, the model enhances its reasoning skills and dedicates more thinking effort to solving problems compared to other baselines. Although trained on a single task, SWE-RL enables the model to generalize its reasoning capabilities across various domains. In contrast, the SFT version, on average, underperforms relative to the original model.

Overall, our results suggest for the first time that reinforcement learning on real-world software data like PRs enables the model to acquire generalized reasoning skills, whereas supervised finetuning steers the language model towards a specific task distribution, leading to performance declines on tasks with lower emphasis, even when a meticulously curated data mix is used.

3.6 Reward ablation

According to Equation (1), the reward design of SWE-RL allows different instantiations of the *compare* function. Throughout the paper, we adopt the sequence similarity between the predicted and the oracle patch, which is a continuous value from 0 to 1. We denote this type of reward as continuous. It is natural to compare this continuous reward with a discrete reward, where the *compare* function outputs 1 if the predicted and oracle patches exactly match each other, and 0 otherwise. We trained a variant of Llama3-SWE-RL with the discrete reward function, using the same training setup as in the continuous reward case.

Reward type	Correct format	Repair (oracle)
Discrete	94.2%	29.0
Continuous	**95.6%**	**34.8**

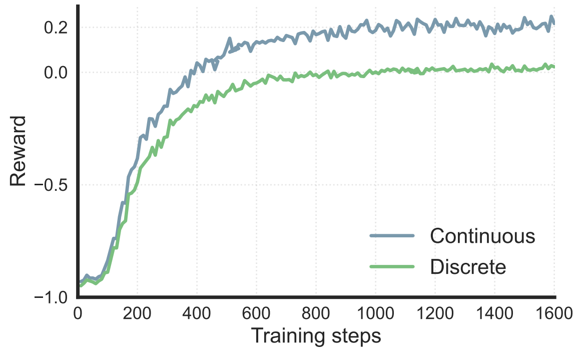


Figure 6 Ablation on SWE-RL's reward functions and their training dynamics. We compare SWE-RL using the default continuous reward function against a discrete reward. Repair (oracle) evaluates the repair-only performance using greedy decoding, with oracle files in the input context.

As shown in Figure 6, while the discrete and continuous reward functions lead to similar format accuracy, the continuous reward is more effective in enhancing the repair performance. From the training dynamics, we can see that discrete rewards grow slower than continuous rewards. Additionally, the average discrete reward remains approximately zero upon the completion of training, meaning it struggles to obtain patches exactly matching the oracles. This is because real-world patches are highly diverse and often cannot be easily matched. The continuous reward function better captures partial correctness and incremental improvements, allowing the model to learn more nuanced and effective repair strategies.

4 Related work

4.1 Language models for software engineering

Large language models (LLMs), trained with billions to trillions of code tokens, have demonstrated outstanding performance in a wide range of coding tasks, including code generation (Chen et al., 2021; Li et al., 2023; Rozière et al., 2023; Guo et al., 2024; Wei et al., 2024; Lozhkov et al., 2024; DeepSeek-AI et al., 2024), code optimization (Cummins et al., 2024; Liu et al., 2024a), program repair (Xia and Zhang, 2022; Xia et al., 2023b; Jiang et al., 2023; Wei et al., 2023), and software testing (Xia et al., 2023a; Deng et al., 2023; Yuan et al., 2023; Schäfer et al., 2023; Lemieux et al., 2023). Initially, researchers primarily focused on single-shot code generation tasks, such as function-level (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021a; Li et al., 2022; Jain et al., 2024b), class-level (Du et al., 2023), and repository-level code completion (Ding et al., 2023; Liu et al., 2024b; Zhang et al., 2023). However, with the rapid development of LLMs, the performance on many popular single-shot code generation benchmarks like HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and EvalPlus (Liu et al., 2023) has become saturated. Since the development of SWE-bench (Jimenez et al., 2023), which requires solving real-world GitHub issues, researchers start to work on improving LLMs' real-world issue-solving capability and have designed various scaffolds for SWE-bench. Two general types are (1) agentic scaffolds (Yang et al., 2024b; Wang et al., 2024; Gauthier, 2024), where an LLM drives the decision-making process based on its past actions and observations through tool-based interaction with the environment; and (2) pipeline-based scaffolds (Xia et al., 2024; Örwall, 2024; Zhang et al., 2024), where an LLM goes through human-defined stages to solve a given issue. Generally, agentic methods are more general but require strong instruction-following and capable LLMs to drive the autonomous process, and can be computationally intensive due to multi-round interactions. In contrast, pipeline-based approaches are more specialized but efficient, with a focus on LLMs' pure code editing capability. Therefore, we designed our minimalist pipeline-based scaffold, Agentless Mini, to focus on the enhancements of Llama3-SWE-RL's core code editing capabiltiy.

4.2 Training software agents

While existing scaffolds have successfully leveraged proprietary language models to tackle real-world software engineering tasks, open models typically yield subpar results in these settings. Moreover, the most effective approach to enhancing real-world software engineering capabilities through training remains unclear. Recently, researchers have begun exploring the possibility of training open LLMs specifically for software engineering tasks, aiming to improve performance on benchmarks such as SWE-bench. For instance, Lingma-SWE-GPT (Ma et al., 2024) introduces 7B and 72B model variants that build on top of Qwen2.5-Coder-7B (Hui et al., 2024) and Qwen2.5-72B-Instruct (Yang et al., 2024a), using an iterative development-process-centric approach. SWE-Gym (Pan et al., 2024) presents the first open training environment for software engineering agents, significantly improving the performance of Qwen2.5-Coder's 7B and 32B variants on SWE-bench. More recently, SWE-Fixer (Xie et al., 2025) finetunes the Qwen2.5 base series, resulting in a 7B code retriever and a 72B code editor focused on efficient issue resolution, achieving notable best@1 improvements. Notably, all these works incorporate distilled samples from either GPT-4o or Claude-3.5-Sonnet in their training data and are built upon Qwen2.5 models. Their training objectives are all based on supervised finetuning. On the contrary, Llama3-SWE-RL is based on Llama 3 (Dubey et al., 2024) and trained through reinforcement learning (RL) using SWE-RL. The seed dataset for RL is sourced exclusively from publicly available repositories, allowing Llama3-SWE-RL to self-improve its issue-solving capabilities through the RL inscentive. Remarkably, Llama3-SWE-RL achieves the best performance among these models with a 41.0% solve rate on SWE-bench Verified (OpenAI, 2024), demonstrating for the first time that LLMs can already effectively address real-world issues through RL on real-world software artifacts.

5 Conclusion

We introduce SWE-RL, the first reinforcement learning (RL) approach to improve language models (LLMs) on software engineering tasks using software evolution data (e.g., PRs) and rule-based rewards. The resulting model, Llama3-SWE-RL-70B, achieves a **41.0%** solve rate on SWE-bench Verified, a human-verified collection of high-quality GitHub issues. This performance is state-of-the-art among medium-sized models and comparable to many proprietary LLMs such as GPT-4o. While SWE-RL is specifically applied to the issue-solving task, Llama3-SWE-RL has developed generalized reasoning skills through RL, demonstrating improved performance on out-of-domain tasks such as code reasoning, mathematics, and general language understanding. Overall, SWE-RL opens a new direction for enhancing the software engineering capabilities of LLMs through RL.

Limitations. Despite the promising results, our approach has several limitations. First, our reward implementation compares the sequence similarity between the predicted and oracle patch rather than their semantic equivalence. This may prevent the policy LLM from exploring alternative, functional equivalent solutions. Additionally, in Agentless Mini, the localization process is simplified to mapping repository structures to file paths, which lacks comprehensive context. Moreover, as a pipeline-based approach, Agentless Mini divides all steps into distinct inference stages. This "external structure" prevents the model from learning through interaction feedback and hinders its ability to consider the entire problem holistically. Furthermore, our approach requires a substantial sampling budget to achieve optimal results, which may be impractical for projects with high execution costs.

Future work. In the future, we plan to address these limitations by integrating agentic reinforcement learning into our framework. This will enable the model to independently learn localization and repair strategies without relying on external structures. Additionally, we will incorporate execution to allow the model to interact directly with the repository environment. We will also focus on improving the sample efficiency during inference, with the goal of achieving equal or better performance with fewer samples. Ultimately, we aim to enhance the practicality of SWE-RL, paving the way for more powerful, reliable, and fully open-source solutions for active GitHub issue resolution.

Acknowledgement

We thank Chris Waterson, Chris Cummins, and Volker Seeker for their assistance in debugging and testing Agentless Mini; Kunhao Zheng, David Zhang, and Taco Cohen for their assistance in the online RL infra; Pierre Chambon, Mihir Sanjay Kale, Parth Thakkar, Michael Jiang, Sinong Wang, and Jingyue Shen for their involvement in the PR dataset discussion; Jannik Kossen for his helpful discussions in model training; Megan Miller, Hannah Schubloom, and Madeline Hepler for their support in the review process; Steven Xia and Yinlin Deng for their aid in setting up Agentless; Jiawei Liu, Yifeng Ding, Terry Yue Zhuo, and Naman Jain for their valuable discussions; Albert Örwall for their help in resolving issues with Moatless EvalTools.

References

Anthropic. Claude 3.5 sonnet model card addendum. *Claude-3.5 Model Card*, 2024a.

Anthropic. Raising the bar on SWE-bench Verified with Claude 3.5 Sonnet. https://www.anthropic.com/research/swe-bench-sonnet, 2024b.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. https://arxiv.org/abs/2108.07732.

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

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code, 2021.

Chris Cummins, Volker Seeker, Dejan Grubisic, Baptiste Roziere, Jonas Gehring, Gabriel Synnaeve, and Hugh Leather. Meta large language model compiler: Foundation models of compiler optimization, 2024. https://arxiv.org/abs/2407.02524.

DeepSeek-AI. Deepseek-v3 technical report, 2024. https://arxiv.org/abs/2412.19437.

DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. https://arxiv.org/abs/2501.12948.

DeepSeek-AI, Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y. Wu, Yukun Li, Huazuo Gao, Shirong Ma, Wangding Zeng, Xiao Bi, Zihui Gu, Hanwei Xu, Damai Dai, Kai Dong, Liyue Zhang, Yishi Piao, Zhibin Gou, Zhenda Xie, Zhewen Hao, Bingxuan Wang, Junxiao Song, Deli Chen, Xin Xie, Kang Guan, Yuxiang You, Aixin Liu, Qiushi Du, Wenjun Gao, Xuan Lu, Qinyu Chen, Yaohui Wang, Chengqi Deng, Jiashi Li, Chenggang Zhao, Chong Ruan, Fuli Luo, and Wenfeng Liang. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence, 2024. https://arxiv.org/abs/2406.11931.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models, 2023.

Hantian Ding, Zijian Wang, Giovanni Paolini, Varun Kumar, Anoop Deoras, Dan Roth, and Stefano Soatto. Fewer truncations improve language modeling. In *Forty-first International Conference on Machine Learning*, 2024. https://openreview.net/forum?id=kRxCDDFNpp.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. https://openreview.net/forum?id=wgDcbBMSfh.

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

AI@Meta: Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, and Angela Fan et al. The llama 3 herd of models, 2024. https://arxiv.org/abs/2407.21783.

Paul Gauthier. Aider is ai pair programming in your terminal. https://aider.chat/, 2024.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning, 2025. https://arxiv.org/abs/2410.02089.

GitHub. Github rest api documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28, 2022. Accessed: 2025-02-24.

GitHub. Github. https://github.com, 2025.

Ilya Grigorik. Gh archive. https://www.gharchive.org/, 2025. Accessed: 2025-02-23.

Alex Gu, Baptiste Roziere, Hugh James Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *Proceedings of the 41st ICML*, volume 235 of *Proceedings of Machine Learning Research*, pages 16568–16621. PMLR, 21–27 Jul 2024. https://proceedings.mlr.press/v235/gu24c.html.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021a.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021b. https://openreview.net/forum?id=d7KBjmI3GmQ.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 2)*, 2021c. https://openreview.net/forum?id=7Bywt2mQsCe.

Shengding Hu, Yuge Tu, Xu Han, Ganqu Cui, Chaoqun He, Weilin Zhao, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Xinrong Zhang, Zhen Leng Thai, Chongyi Wang, Yuan Yao, Chenyang Zhao, Jie Zhou, Jie Cai, Zhongwu Zhai, Ning Ding, Chao Jia, Guoyang Zeng, dahai li, Zhiyuan Liu, and Maosong Sun. MiniCPM: Unveiling the potential of small language models with scalable training strategies. In *First Conference on Language Modeling*, 2024. https://openreview.net/forum?id=3X2L2TFr0f.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-coder technical report, 2024. https://arxiv.org/abs/2409.12186.

Kush Jain, Gabriel Synnaeve, and Baptiste Rozière. Testgeneval: A real world unit test generation and test completion benchmark, 2024a. https://arxiv.org/abs/2410.00752.

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

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues?, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench leaderboard. https://www.swebench.com, 2024. Accessed: 2025-02-04.

Caroline Lemieux, Jeevana Priya Inala, Shuvendu K Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 919–931. IEEE, 2023.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy, Jason Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan

Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you!, 2023.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, December 2022. ISSN 1095-9203. doi: 10.1126/science.abq1158. http://dx.doi.org/10.1126/science.abq1158.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. https://openreview.net/forum?id=1qvx610Cu7.

Jiawei Liu, Songrun Xie, Junhao Wang, Yuxiang Wei, Yifeng Ding, and Lingming Zhang. Evaluating language models for efficient code generation. In *First Conference on Language Modeling*, 2024a. https://openreview.net/forum?id=IBCBMeAhmC.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024b. https://openreview.net/forum?id=pPjZIOuQuF.

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

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, 2024. https://arxiv.org/abs/2411.00622.

OpenAI. Gpt-4o system card, 2024a. https://arxiv.org/abs/2410.21276.

OpenAI. Openai o1 system card, 2024b. https://arxiv.org/abs/2412.16720.

OpenAI. simple-evals. https://github.com/openai/simple-evals, 2024. Accessed: 2025-02-23.

OpenAI. Introducing swe-bench verified. https://openai.com/index/introducing-swe-bench-verified, 2024. Accessed: 2025-02-04.

Jiayi Pan, Xingyao Wang, Graham Neubig, Navdeep Jaitly, Heng Ji, Alane Suhr, and Yizhe Zhang. Training software engineering agents and verifiers with swe-gym, 2024. https://arxiv.org/abs/2412.21139.

John W. Ratcliff and David E. Metzener. Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal*, page 46, July 1988. https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code, 2023.

Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation. *IEEE Transactions on Software Engineering*, 2023.

John Schulman. Approximating kl divergence. http://joschu.net/blog/kl-approx.html, 2020. Accessed: 2025-02-22.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024. https://arxiv.org/abs/2402.03300.

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, 2024. https://arxiv.org/abs/2407.16741.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. https://openreview.net/forum?id=_VjQlMeSB_J.

Yuxiang Wei, Chunqiu Steven Xia, and Lingming Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair, 2023.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with OSS-instruct. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 52632–52657. PMLR, 21–27 Jul 2024. https://proceedings.mlr.press/v235/wei24h.html.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning, 2022.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models, 2023a.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1482–1494, 2023b. doi: 10.1109/ICSE48619.2023.00129.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. https://arxiv.org/abs/2407.01489.

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, 2025.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, Kai Dang, Keming Lu, Keqin Bao, Kexin Yang, Le Yu, Mei Li, Mingfeng Xue, Pei Zhang, Qin Zhu, Rui Men, Runji Lin, Tianhao Li, Tingyu Xia, Xingzhang Ren, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yu Wan, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, and Zihan Qiu. Qwen2.5 technical report, 2024a.

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*, 2024b. https://openreview.net/forum?id=mXpq6ut8J3.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024c. https://arxiv.org/abs/2410.03859.

Edward Yeo, Yuxuan Tong, Morry Niu, Graham Neubig, and Xiang Yue. Demystifying long chain-of-thought reasoning in llms, 2025. https://arxiv.org/abs/2502.03373.

Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207*, 2023.

Huaye Zeng, Dongfu Jiang, Haozhe Wang, Ping Nie, Xiaotong Chen, and Wenhu Chen. Acecoder: Acing coder rl via automated test-case synthesis. *ArXiv*, 2502.01718, 2025.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. https://arxiv.org/abs/2404.05427.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024. https://arxiv.org/abs/2412.01769.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Albert Örwall. Moatless tools. https://github.com/aorwall/moatless-tools, 2024.

Appendix

A Agentless Mini

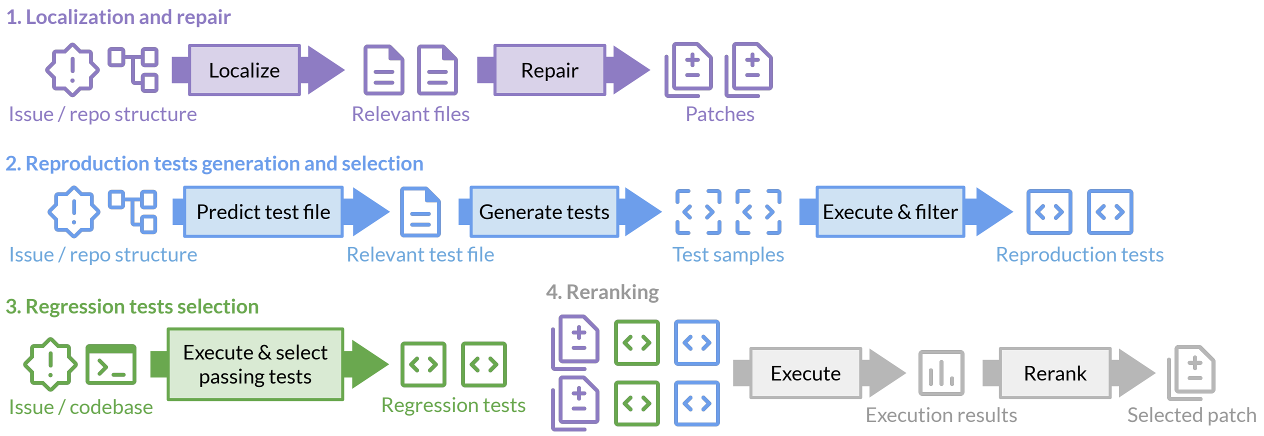


Figure 7 The Agentless Mini scaffold. The design emphasizes easy decomposition, parallelization, and scalability.

In addition to a model proficient in code editing, effectively tackling software engineering tasks, such as those found in SWE-bench (Jimenez et al., 2023), also requires a robust scaffold. Agentless (Xia et al., 2024) is one of the state-of-the-art scaffolds for SWE-bench at the time of writing. Building upon Agentless with various simplifications and enhancements, we developed Agentless Mini, a framework that prioritizes straightforward component decomposition, parallelization, and scalability. With Agentless Mini, each step's inference or execution compute can be independently scaled to enhance SWE-bench performance. In Figure 7, we present a detailed illustration of Agentless Mini's working principles. The following paragraphs will elaborate on each step and highlight the differences from Agentless.

Localization and repair. For localization, we employ a prompting-based approach that enables the model to predict relevant file paths based on a given issue and the repository's structure. Unlike Agentless, which involves two additional detailed steps to identify related elements and files, as well as a separate embedding model, Agentless Mini simplifies the process. It generates multiple samples of potentially problematic files from the model and consolidates them into unique sets for repair.

During the repair phase, the LLM is conditioned on the full content of the files to predict search/replace edits. We generate multiple repair samples from different location sets, ensuring a comprehensive exploration of the patch search space.

Reproduction tests generation and selection. Agentless samples reproduction tests for patch selection. Initially, multiple reproduction tests are generated based on an issue description, and one majority sample is selected after filtering. These tests must have distinct logic to output `"Issue reproduced"` and `"Issue resolved"` when the issue is reproduced or resolved, respectively. They are filtered based on whether they correctly output `"Issue reproduced"` when executed in the original codebase. Agentless Mini enhances this pipeline with two key improvements. First, instead of relying solely on the issue description, the model retrieves a relevant test file to guide test generation. Additionally, rather than selecting just one majority sample, Agentless Mini allows for the selection of multiple top test samples based on voting results. In our evaluation, using more test samples has proven beneficial for reranking (§3.4).

Regression tests selection. We select regression tests in the same manner as Agentless. Initially, we gather all passing tests for each issue by executing the code before any modifications. This step does not require model inference and needs to be performed only once. Subsequently, an additional, optional inference step is conducted to filter out passing tests that are supposed to fail after the issue is fixed. The rest of the tests will be marked as regression tests.

Reranking. Agentless Mini utilizes both regression and reproduction tests for reranking. For each issue, every applied patch is executed against the regression tests. The patches that result in the minimum number of existing test failures are selected. For each issue, we choose the top-N reproduction tests and run each patch against these tests. If the execution outputs `"Issue resolved"`, we mark the patch passing this test. We then adopt the dual execution agreement objective from CodeT (Chen et al., 2023). Specifically, patches $\mathcal{P}$ that pass the same set of reproduction tests $\mathcal{T}$ are denoted as a consensus group. Each consensus group is scored using the formula $|\mathcal{P}| \times |\mathcal{T}|^2$. With this objective, consensus groups with patches passing more tests receive higher scores. Additionally, groups where more patches pass the tests are scored higher, although passing more tests is prioritized over having more patches. Finally, we identify the consensus group with the highest score and select the best patch from this group using majority voting.

B Synthesizing supervised-finetuning data

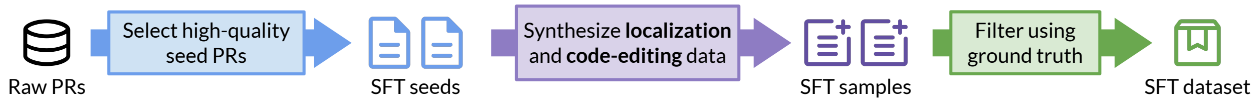

Raw PRs
Select high-quality seed PRs
SFT seeds
Synthesize localization and code-editing data
SFT samples
Filter using ground truth
SFT dataset

Figure 8 Synthetic data pipeline for constructing SFT data. We start by collecting high-quality seed PRs using heuristics, then generate synthetic localization and code-editing samples, and finally use the ground-truth edited files and patches to filter out incorrect samples.

Figure 8 shows our method of generating synthetic supervised-finetuning (SFT) data. The data generation pipeline is inspired by Magicoder (Wei et al., 2024), where the OSS-Instruct technique generates high-quality code instruction data from open-source seed snippets. We apply a similar methodology to generate both fault localization and code editing data using high-quality PR seeds.

Collecting high-quality seed PRs. To begin with, we extract high-quality PR seeds from the raw dataset we collected, as detailed in §2.1. These seeds are chosen based on specific heuristics. For example, a PR instance should include at least one linked issue, the issue should describe a bug fix request, and the code changes should involve programming files.

Localization and editing data synthesis. We adopt Llama-3.3-70B-Instruct (Dubey et al., 2024) for data synthesis. For localization data, we prompt the model with the issue description, repository structure, and the paths of edited and relevant files as hints. We then ask the model to identify the relevant files for modification or review by generating a thought process, followed by a prioritized list of file paths. During filtering, we ensure that the model's response includes all files that are genuinely edited in the PR, and these files should be prioritized in the ranking.

Similarly, in terms of code editing, we synthesize code edits for a given issue, in search/replace format (Xia et al., 2024), by providing the ground-truth PR and patch as guidance to Llama-3.3-70B-Instruct. During the filtering process, we ensure that all search/replace blocks adhere to the correct format and that all search paths and blocks can be accurately matched within the input files' context.

SFT baseline. As discussed in §3.1, we meticulously constructed Llama3-SWE-SFT-70B as a strong SFT baseline. This SFT baseline is trained on a 16k context window for 2B tokens, where the training data consists of a mix of the aforementioned synthetic localization and editing data, as well as coding and general SFT datasets from Llama 3 (Dubey et al., 2024).

C Midtraining on large-scale PR data

In addition to our main technique, SWE-RL, we are exploring orthogonal ways to improve language models (LLMs) on real-world software engineering. In this section, we explain how we can directly utilize the raw PR collection through continued pretraining (midtraining) to empower small LLMs, such as Llama-3.1-8B, to achieve a high resolve rate (**31.0%**) on SWE-bench Verified (OpenAI, 2024).

Data packing design. In pretraining, documents are often concatenated into a long sequence and split by the context window, avoiding padding and boosting efficiency. However, Ding et al. (2024) highlights that truncations can cause issues like hallucinations because a single document can be split into different training sequences. Since midtraining involves much fewer tokens than pretraining, we choose to enhance the representation accuracy and sacrafice some training efficiency. As shown in Figure 9, we truncate PR documents that exceed the context window. For shorter documents, we pack several together in a sequence, using padding tokens as needed. Attention masks are applied to block attention between documents in the same sequence. The same packing approach is used in posttraining with SFT. Approximately, this packing strategy leads to a 75% non-padding rate for each batch.

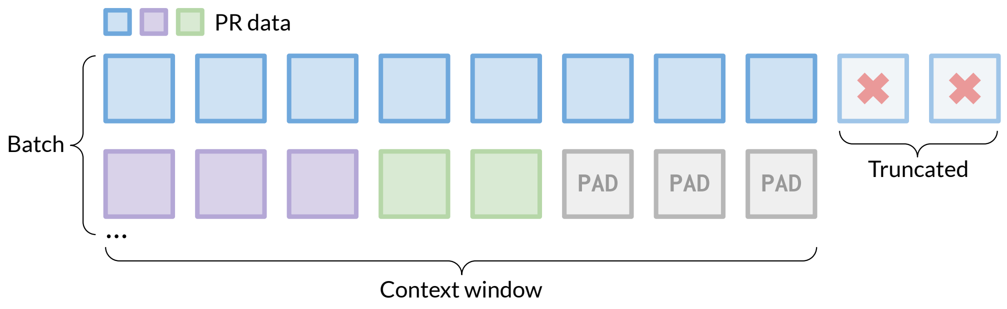


Figure 9 Data packing design for midtraining. Documents longer than the context window are truncated at the end, while shorter documents are packed into a single sequence along with padding.

Formatting PR data as dialogs. We format each mid-training PR instance as a dialogue between the user and the assistant, with the user's content (e.g., code context) being masked. This approach is beneficial for the following reasons: (1) loss masking prevents the repetition of code context across different PRs of the same repository, (2) masking helps ensure that the model focuses on predicting code edits rather than overly concentrating on generating code context, which usually occupies most of the context window, and (3) maintaining a consistent dialogue format with posttraining can facilitate smoother model learning and allows us to reuse the training infrastructure effectively.

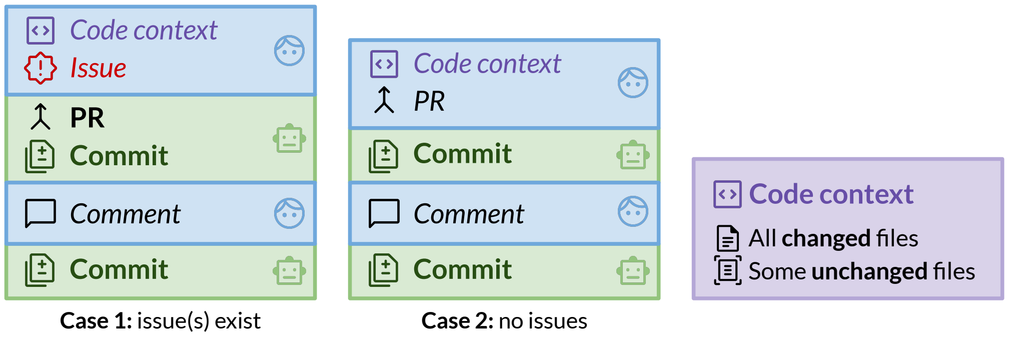


Figure 10 Formatting midtraining PR data as dialogs. Code context includes contents of all changed files and some unchanged but relevant files, PR denotes a PR title and description, Commit indicates consecutive commits with messages and code changes, and Comment indicates consecutive GitHub user conversations or review comments. Commits and Comments are ordered chronologically.

Figure 10 illustrates the details of how we format each PR instance. The code context is always included in the first user message, which includes content of all changed files and some unchanged but relevant files. We further examine two scenarios based on whether the PR references any issues. In cases where issues are present, we include both the code context and the issue details in the initial user dialog. The PR description and the first series of consecutive commits are then provided in the assistant's initial response. This approach encourages the model to learn the entire issue resolution process by predicting the PR and its associated commits. Conversely, when no issues are mentioned, we present the PR in the user's initial input, allowing the model to learn how to implement code changes based on the PR description. The remaining dialog turns are divided based on the timing of the subsequent commits. Commits serve as turning points, represented by assistant responses, while other comments are represented as user inputs. The unified diff of each commit are randomly converted into LLM-friendly editing format such as search/replace (Xia et al., 2024).

Dynamic context adjustment. Although including code context in the dialog is essential for the model to learn how to process contextual information, the context can often be excessively long and exceed the context window, meaning that the entire sequence would not contribute to any actual loss during training. From our observation, a large number of data points would be invalid without additional processing. To address this issue, we implement *dynamic context adjustment*. The goal of this strategy is to shorten the code context while preserving the most relevant information, particularly the edited parts. We achieve this by randomly replacing segments of code with an ellipsis ("...") to reduce the overall length of the context. We apply removal of code segments uniformly across all files to prevent bias. The amount of removal is determined by a parameter indicating the expected context size. In our implementation, we estimate the token count using string length to efficiently apply the pipeline without requiring explicit tokenization. Overall, this strategy allows us to keep most of the PR documents.

Stable training and annealing. We divide the midtraining process into two stages: a stable training stage and an annealing stage, as outlined in MiniCPM (Hu et al., 2024) and Llama 3 (Dubey et al., 2024). We utilize a trapezoidal learning rate scheduler, also known as Warmup-Stable-Decay (Hu et al., 2024). In this approach, the learning rate remains constant during the stable training stage after an initial linear warmup, followed by a linear decay during the annealing stage. During the stable training stage, we exclusively use midtraining data. In the annealing stage, we incorporate additional datasets, including the synthetic SFT data (Appendix B) for localization and editing, as well as some coding data and general-purpose dialogs from Llama 3.

Llama3-Midtrain-8B (beta). Llama3-Midtrain-8B (beta) is midtrained from the base Llama-3.1-8B (Dubey et al., 2024) for 500B tokens using a 32k context window. We perform annealing for the last 10% of the midtraining steps, where we mix a small fraction of our posttraining SFT data (Appendix B), along with coding and general dialog data from Llama 3 (Dubey et al., 2024). It is then SFT-ed for 8B tokens using a 64k context window. The resulting model achieves a **31.0%** resolve rate on SWE-bench Verified, significantly outperforming previous small LLM baselines and comparable to many results achieved with larger models. However, we have observed some limitations, such as a reduction in general capabilities due to this large-scale midtraining. Looking ahead, we aim to overcome this challenge by developing a more refined data mix to enhance the model's overall performance.

D Complete prompt

The following is a complete version of Figure 3:

```
PROMPT_TEMPLATE = """<|begin_of_text|><|start_header_id|>system<|end_header_id|>

A user will ask you to solve a task. You should first draft your thinking process (inner
    monologue). Then, generate the solution.

Your response format must follow the template below:
<think>
Your thoughts or/and draft, like working through an exercise on scratch paper. Be as
    casual and as long as you want until you are confident to generate a correct solution.
</think>
<solution>
Final solution presented to the user.
</solution><|eot_id|><|start_header_id|>user<|end_header_id|>

We are currently solving the following issue within our repository. Here is the issue
    text:
--- BEGIN ISSUE ---
{problem_statement}
--- END ISSUE ---
```

~~~
Below are some code segments, each from a relevant file. One or more of these files may contain bugs.

--- BEGIN FILE ---
```
{content}
```
--- END FILE ---

Please first localize the bug based on the issue statement, and then generate *SEARCH/REPLACE* edits to fix the issue.

Every *SEARCH/REPLACE* edit must use this format:
1. The file path
2. The start of search block: <<<<<<< SEARCH
3. A contiguous chunk of lines to search for in the existing source code
4. The dividing line: =======
5. The lines to replace into the source code
6. The end of the replace block: >>>>>>> REPLACE

Here is an example:

```python
### mathweb/flask/app.py
<<<<<<< SEARCH
from flask import Flask
=======
import math
from flask import Flask
>>>>>>> REPLACE
```

Please note that the *SEARCH/REPLACE* edit REQUIRES PROPER INDENTATION. If you would like to add the line '        print(x)', you must fully write that out, with all those spaces before the code!
Wrap each *SEARCH/REPLACE* edit in a code block as shown in the example above. If you have multiple *SEARCH/REPLACE* edits, use a separate code block for each one.<|eot_id|><|start_header_id|>assistant<|end_header_id|>

"""
~~~