SoRFT: Issue Resolving with Subtask-oriented Reinforced Fine-Tuning

Zexiong Ma[*] [♢], **Chao Peng**[†][♣], **Pengfei Gao** [♣],**Xiangxin Meng** [♣], **Yanzhen Zou**[♢], **Bing Xie**[†][♢]

[♢]School of Computer Science, Peking University, [♣]ByteDance, Beijing

[♢]mazexiong@stu.pku.edu.cn, {zouyz, xiebing}@pku.edu.cn,

[♣]{pengchao.x, gaopengfei.se, mengxiangxin.1219}@bytedance.com

Abstract

Mainstream issue-resolving frameworks predominantly rely on commercial models, leading to high costs and privacy concerns. Existing training approaches for issue resolving struggle with poor generalization and fail to fully leverage open-source development resources. We propose **S**ubtask-**o**riented **R**einforced **F**ine-**T**uning (**SoRFT**), a novel training approach to enhance the issue resolving capability of LLMs. We decomposes issue resolving into structured subtasks: file localization, function localization, line localization, and code edit generation. SoRFT consists of two training stages: (1) **rejection-sampled supervised fine-tuning**, Chain of Thought (CoT) data is filtered using ground-truth before fine-tuning the LLM, and (2) **rule-based reinforcement learning**, which leverages PPO with ground-truth based rewards. We evaluate the SoRFT-trained model on SWE-Bench Verified and SWE-Bench Lite, achieving state-of-the-art (SOTA) performance among open-source models (e.g., resolve 21.4% issues on SWE-Bench Verified with SoRFT-Qwen-7B). The experimental results demonstrate that SoRFT significantly enhances issue-resolving performance, improves model generalization, and provides a cost-efficient alternative to commercial models.

1 Introduction

Large Language Models (LLMs)(OpenAI, 2023; Touvron et al., 2023) have demonstrated exceptional performance across a wide range of complex real-world tasks (Li et al., 2022, 2024; Wu et al., 2023), particularly excelling in software development (Jimenez et al., 2024; Zhao et al., 2024; Ma et al., 2024c). The current mainstream automated software development systems mainly use commercial models (OpenAI, 2023; Cla, 2024). However, the API call of commercial models are costly and pose privacy leakage issues, limiting their application in development processes in the industry.

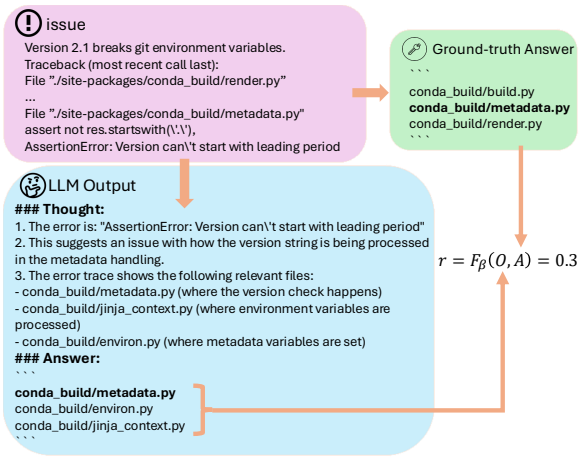

Figure 1: Rule-based reward example for file localization subtask. LLM generates CoT data for a given issue, the reward for the sampled CoT is then calculated by the F_β score based on the extracted answer and the ground-truth answer.

Research communities have attempted to fine-tune open-source LLMs (Team, 2024a; Guo et al., 2024; Cod, 2024; Touvron et al., 2023; Hui et al., 2024; Lozhkov et al., 2024) to improve their performance in Issue Resolving task. Existing Approaches(Pan et al., 2024; Ma et al., 2024b; Xie et al., 2025a; Ma et al., 2024a) utilize LLMs to sample Chain-of-Thought (CoT)(Wei et al., 2022) data, then perform Negative Sample Filtering based on ground-truth data, and fine-tune the models. While these methods improve LLMs' issue-resolving capabilities, relying solely on supervised fine-tuning (SFT) can lead to poor generalization, making models more susceptible to hallucinations and factual errors.

Recent studies (Luong et al., 2024; Guo et al., 2025; Zeng et al., 2025; Xie et al., 2025b; Mu et al., 2024; Team et al., 2025) have explored rule-based reinforcement learning to enhance model performance on complex tasks such as mathematics.

[*] Work done during an internship at ByteDance.

[†] Corresponding authors.

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 11427–11441

Rule-based reinforcement learning requires ground-truth for evaluation, but constructing ground-truth for math problems is labor-intensive. In the open-source community, a vast number of resolved issues come with ground-truth patches. This raises a natural question: *Can we leverage these (issue, patch) pairs for rule-based reinforcement learning to improve the issue-resolving capabilities of language models?*

In this paper, we propose **S**ubtask-**o**riented **R**einforced **F**ine-**T**uning (**SoRFT**), fully utilizes both positive and negative sample information to improve model performance in Issue Resolving. Given the complexity of the Issue Resolving task (Jimenez et al., 2024), constructing end-to-end training data is challenging. Inspired by Agentless (Xia et al., 2024), we break down issue resolving into multiple subtasks: file localization, function localization, line localization, and code edit generation—and derive ground-truth answers for each subtask from the pull requests associated with the issues. We then perform Reinforced Fine-Tuning (Luong et al., 2024) for these subtasks.

SoRFT consists of two training stages: **rejection-sampled supervised fine-tuning** (SFT) and **rule-based reinforcement learning** (RL). In the SFT stage, we employ a teacher LLM to generate CoT data for each subtask and filter negative samples based on ground-truth answers. We then perform supervised fine-tuning to help the model grasp the subtask structures, underlying reasoning mechanisms, and output formats essential for issue resolving. In the RL stage, since each subtask has a corresponding ground-truth, we employ rule-based proximal policy optimization (PPO) (Schulman et al., 2017) for training. Specifically, we define scoring rules based on ground-truth for each subtask, and update the LLM's parameters using reward-based optimization. This process further improves the model's issue-resolving performance. SoRFT-trained LLMs achieve state-of-the-art performance on SWE-Bench Verified and SWE-Bench Lite, demonstrating the effectiveness of SoRFT in enhancing issue-resolving capabilities. In this paper, we make the following contributions:

- We introduce Subtask-oriented Reinforced Fine-Tuning (SoRFT), designing rule-based rewards for each issue-resolving subtask and enhancing LLMs' issue-resolving capabilities through reinforced fine-tuning.
- We apply SoRFT to open-source models and validate its effectiveness within Agentless framework.
- We investigate the impact of different reward rules on PPO training, providing insights for designing more robust reward rules.

2 Background

In this section, we provide a brief introduction to SWE-Bench, issue resolving framework, and the reinforcement learning algorithm.

2.1 SWE-Bench

SWE-Bench (Jimenez et al., 2024) is a benchmark to evaluate language models' ability to resolve real-world software issues, such as bug reports and feature requests on GitHub. LLM-based programming assistants are given an issue description along with the entire repository and are expected to generate code edits that resolve the issue. SWE-Bench Lite (Team, 2024b) is a curated subset of SWE-Bench, specifically focus on evaluating functional bug fixes. While SWE-Bench Verified (OpenAI, 2024c) is a human-verified subset addressing quality issues in SWE-Bench, such as vague problem descriptions. All issue-resolving experiments in this paper are conducted on SWE-Bench Lite and SWE-Bench Verified.

2.2 Issue Resolving Framework

Issue resolving frameworks can be broadly divided into two categories: agent-based and pipeline-based. Openhands(Wang et al., 2024b) is a purely react-style (Yao et al., 2022) agent-based framework for software development tasks. Xie et al. (2025a) propose SWE-Fixer, a two-stage pipeline-based system. Ma et al. (2024a) propose SWE-SynInfer, a hybrid framework that combines pipeline design with agent-based approaches. Additionally, Xia et al. (2024) propose Agentless, a multi-stage pipeline-based framework designed to fully leverage the reasoning capabilities of LLMs for issue resolving. Notably, Agentless is the state-of-the-art (SOTA) pipeline-based framework on the SWE-Bench leaderboard and has been adopted by OpenAI (OpenAI, 2024b) and DeepSeek (Guo et al., 2025) to assess their models' performance in software engineering tasks. Constructing training data requires sampling CoT data for the framework and filtering out negative samples. Assessing the accuracy of intermediate steps in the Agent framework is challenging, whereas the pipeline-based

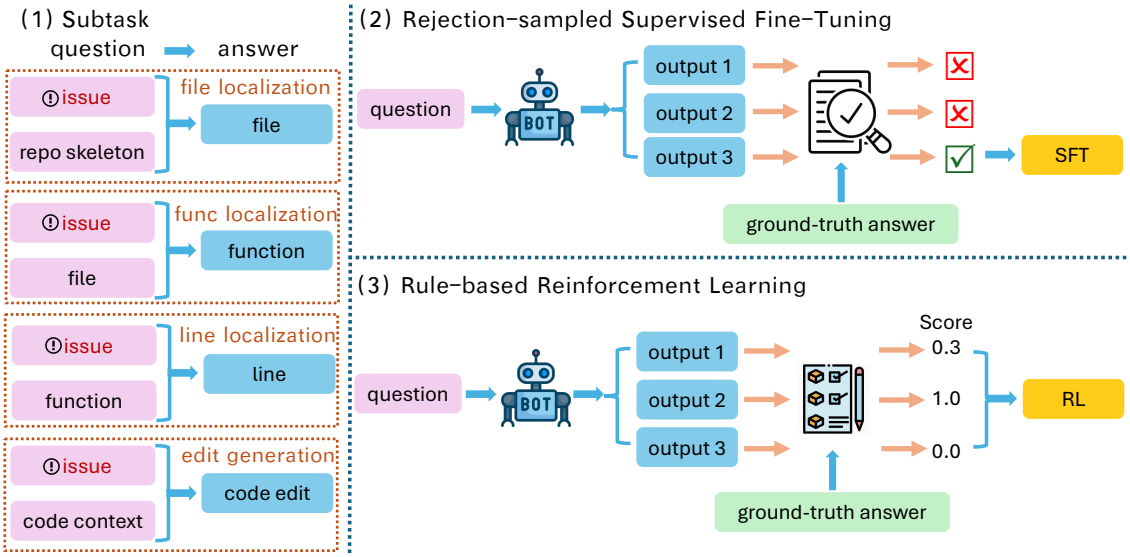

Figure 2: SoRFT consists three parts: (1) decompose issue resolving into four subtasks: file localization, function localization, line localization and code edit generation; (2) fine-tune LLMs with rejection-sampled CoT data to enable it follow the task format and reasoning methods for each subtask; (3) employ rule-based reinforcement learning to further enhance the issue resolving ability of LLMs.

framework offers a clearer and more effective approach to evaluating reasoning accuracy at each stage. In this paper, we design training sub-tasks based on Agentless and employ Agentless[1] as the inference framework in our experiments.

2.3 Reinforcement Learning

Reinforcement learning algorithms are widely used in the alignment phase of large language models (LLMs), including proximal policy optimization (PPO) (Schulman et al., 2017), group relative policy optimization (GRPO) (Shao et al., 2024), direct preference optimization (DPO) (Rafailov et al., 2023), and Kahneman-Tversky optimization (KTO) (Ethayarajh et al., 2023, 2024). The PPO algorithm calculates the reward using Equation (1) and subsequently updates the parameters of the Policy Model.

$$r = r_\phi(q, o) - \beta \cdot \mathrm{KL}[\pi_\theta(\cdot|q) \parallel \pi_{ref}(\cdot|q)] \quad (1)$$

where q represents the given question, o refers to the predicted output, r_ϕ is the reward model, π_θ is the policy model, π_{ref} is the reference model (mostly refer to the original policy model $\pi_{\theta_{old}}$). As PPO is capable of maintaining stability and efficiency during policy updates, we employ PPO for RL training in this paper.

In the standard PPO algorithm (Schulman et al., 2017), a pre-trained reward model is used to score responses. However, recent studies (Guo et al., 2025; Team et al., 2025) have demonstrated that relying on a reward model can lead to reward hacking, and adopting a more objective scoring approach can effectively mitigate this issue. Since ground-truth can be easily collected for issue resolving tasks, we designed a set of scoring rules to evaluate responses specifically for the issue resolving task. We will present the reward rules in Section 3.3.

3 Approach

As shown in Figure 2, SoRFT contains three parts: (1) issue resolving subtask construction, (2) rejection-sampled supervised fine-tuning, and (3) rule-based reinforcement learning.

3.1 Issue Resolving Subtasks

To address the challenge of constructing end-to-end training data for the Issue Resolving task, we create training data for phased subtasks. As shown in Figure 2(1), inspired by the design patterns of advanced issue-resolving frameworks (Xia et al., 2024; Ma et al., 2024a), we decompose issue resolving into four subtasks: file localization, function localization, line localization, and code edit generation. This structured decomposition enables a more targeted and effective training process for each phase of the task.

File Localization. File localization training enhances the LLMs' understanding of the high-level

[1]We employ Agentless 1.0 in our paper.

architecture of the repository, enabling them to perform an initial rough localization of relevant files based on the issue description. We utilize the issue and repository structure as inputs, with the full names of the modified files in the pull request (PR) as outputs. To improve the quality of the training data, we excluded non-Python files and test scripts from both input and output. The relationship can be formulated as follows:

$$\text{prompt}_{\text{file}}(\mathcal{I}, \mathcal{R}_s) \rightarrow \mathcal{F}_g \tag{2}$$

where $\mathcal{F}_g$ represents the golden file, $\mathcal{I}$ represents the issue, $\mathcal{R}_s$ represents the repository skeleton.

Function Localization. Function localization training can improve the LLMs' performance on fine-grained localization based on the functional characteristics of the code. In function localization, the issue and file skeleton composed of function names are employed as inputs, while the names of modified functions from the PRs are employed as outputs. This relationship is expressed as:

$$\text{prompt}_{\text{function}}(\mathcal{I}, \mathcal{F}_{g,s}) \rightarrow \mathcal{FN}_g \tag{3}$$

where $\mathcal{FN}_g$ represents the golden function name, $\mathcal{I}$ represents the issue, $\mathcal{F}_{g,s}$ represents the skeleton of the golden file $\mathcal{F}_g$.

Line Localization. Line localization training enhances the LLMs' ability to precisely identify the exact lines of code that require modification to resolve the issue. Line localization takes the issue description and function content as inputs and outputs the modified lines from the PR. This can be formulated as:

$$\text{prompt}_{\text{line}}(\mathcal{I}, \mathcal{FN}_{g,c}) \rightarrow \mathcal{L}_g \tag{4}$$

where $\mathcal{L}_g$ represents the golden modified line, $\mathcal{I}$ represents the issue, $\mathcal{FN}_{g,c}$ represents the content of the golden function $\mathcal{FN}_g$.

Code Edit Generation. Training code edit generation can enhance the LLMs' ability to modify code snippets based on the issue. The input for code edit consists of the issue and the localized code snippet, while the output is the code edits of the corresponding PR. Following previous work (Xia et al., 2024; Yang et al., 2024), we employ the *Search/Replace* edit format. The *Search/Replace* format consists of two main parts: 1) Search: the target code snippet that need to modify, and 2) Replace: the new code snippet after editing. This relationship can be formulated as:

$$\text{prompt}_{\text{edit}}(\mathcal{I}, \mathcal{C}_g) \rightarrow \mathcal{E}_g \tag{5}$$

where $\mathcal{E}_g$ represents the golden code edit, $\mathcal{I}$ represents the issue, $\mathcal{C}_g$ represents the golden code context.

All subtasks are constructed based on resolved issues from open-source projects, and the ground-truth answers are extracted from the corresponding pull requests. The prompts for the subtasks are provided in Appendix D.

Algorithm 1: Rule-based reward

Input: Subtask type s, question prompt q, ground-truth answer A, LLM output o
Output: Reward score r

```
if s == localization then
    O_loc ← extract_locations(o);
    Q ← extract_locations(q);
    if |O_loc| == 0 or |O_loc − Q| > 0 then
        r = 0.0
    else
        r = F_β(O_loc, A)
if s == edit then
    O_search ← extract_search_blocks(o);
    O_edit ← extract_edits(o);
    Q ← extract_code(q);
    if |O_edit| == 0 or |O_search − Q| > 0 then
        r = 0.0
    else
        r = F_β(O_edit, A)
return r
```

3.2 Rejection-sampled Supervised Fine-Tuning

We fine-tune the LLM using Rejection-sampled CoT data to enhance its understanding of the task format and reasoning process for each subtask. As shown in Figure 2(2), we sample CoT data using the LLM and then filter the CoT data based on the ground-truth answer. Specifically, for the three localization subtasks, we filter out samples that have no overlap with the ground-truth file, function or line. For the code edit generation subtask, we filter out samples that have no overlap with the lines modified by the ground-truth edits. Finally, we integrate CoT data from all subtasks to fine-tune the LLM, enabling it to comprehend both the task format and its underlying reasoning mechanisms.

3.3 Ruled-based Reinforcement Learning

We further enhance the reasoning ability and generalization of LLMs on issue-resolving through Rule-based Reinforcement Learning. As shown in Algorithm 1, we utilize the ground-truth answer to calculate the rule-based reward for each subtask. For the localization subtask (line 1-7), we first extract the localization result O_{loc} from the response. If the localization result is empty or contains a target not present in the problem description, the reward is set to 0; otherwise, the reward is calculated as the F_β score between O_{loc} and the ground-truth answer A. For the code editing subtask (line 8-15), we first extract the modification target O_{search} and the modification code O_{edit} from the response. If the modification code is empty or the modification target does not appear in the problem description, the reward is 0; otherwise, the reward is calculated as the F_β score between O_{edit} and the ground-truth answer A. During reinforcement learning period, we replace the reward model in PPO with above rule-based reward. This approach effectively mitigates the risk of reward hacking, ensuring that the model's learning process is guided by precise and reliable feedback.

$$F_\beta = (1+\beta^2) \cdot \frac{\text{Precision} \cdot \text{Recall}}{(\beta^2 \cdot \text{Precision}) + \text{Recall}}, \quad (6)$$

$$\text{Precision} = \frac{|O \cap A|}{|O|}, \text{Recall} = \frac{|O \cap A|}{|A|} \quad (7)$$

where O represents the outputs generated by the LLMs, A denotes the ground-truth answers for the subtask. β is a hyperparameter that balances the impact of precision and recall on final score. Since recall has a greater influence on the final outcome across different subtasks, β should be a value greater than 1. In our experiments, we set $\beta = 3$ to prioritize recall while maintaining a reasonable trade-off with precision.

4 Experiments

In this section, we will introduce our evaluation benchmark, metrics, models, baselines and implementation details.

4.1 Benchmark

We conduct experiments on two issue resolving benchmarks: SWE-Bench Verified and SWE-Bench Lite.

SWE-Bench Verified (OpenAI, 2024c) is a manually verified subset of SWE-bench (Jimenez et al., 2024) with 500 instances. Each instance is a issue associated with test cases that can be executed in a Docker[2] environment. Issue resolving frameworks (Xia et al., 2024; Xie et al., 2025a; Wang et al., 2024b; Ma et al., 2024a) are asked to understand the issue and the repository, generate patches and pass all test cases, providing a reliable evaluation of their issue resolving capabilities.

SWE-Bench Lite (Team, 2024b) is the subset of SWE-Bench containing 300 instances and focuses on evaluating functional bug fixes.

4.2 Metrics

To evaluate the performance of issue resolving frameworks with SoRFT-trained LLMs, we apply two metrics: **%Resolved**, **%Applied**.

%Resolved is the proportion of samples in which applying the generated code edit successfully passed all test cases.

%Applied is the proportion of samples that issue resolving frameworks successfully generate valid code edits that could be applied to the repositories. This metric evaluates the framework's capability to produce practical and executable solutions.

4.3 Framework and Model

We apply Agentless(Xia et al., 2024) as our issue resolving framework and Qwen2.5-Coder (Hui et al., 2024) series as our base model. Agentless is an advanced open-source issue-resolving framework, used by OpenAI (OpenAI, 2024b) and DeepSeek (Guo et al., 2025) to evaluate model performance on software engineering tasks. For base model, we employ Qwen2.5-Coder-7B-Instruct and Qwen2.5-Coder-32B-Instruct(Hui et al., 2024), which is the SOTA open-source coder instruct models (Touvron et al., 2023; Guo et al., 2024).

4.4 Baselines

Since issue resolving tasks necessitate the use of agent-based or pipeline-based frameworks, existing fine-tuning approaches are typically designed and optimized for specific frameworks. In this work, we evaluate our method against three baselines: (1) OpenHands with SWE-Gym-Qwen, (2) SWE-Fixer with SWE-Fixer-Qwen, and (3) SWE-SynInfer with Lingma-SWE-GPT.

Openhands with SWE-Gym-Qwen(Pan et al., 2024). Openhands(Wang et al., 2024a) is a purely

[2]https://www.docker.com/

Table 1: The %Resolved performance of various models on SWE-Bench Verified and SWE-Bench Lite. Given that all fine-tuning approaches are inherently framework-specific, we compare SoRFT-Qwen with previous fine-tuned models within corresponding frameworks.

Model	Framework	Type	Verified	Lite
Proprietary Models				
Claude-3.5-Sonnet (Cla, 2024)	Openhands	Agent	53.0	41.7
Claude-3.5-Sonnet (Cla, 2024)	Agentless	Pipeline	50.8	40.7
o1-preview (OpenAI, 2024b)	Agentless	Pipeline	41.3	-
DeepSeek-R1-672B (Guo et al., 2025)	Agentless	Pipeline	49.2	-
GPT-4o (OpenAI, 2024a)	SWE-SynInfer	Pipeline + Agent	31.8	20.7
7 - 14B Open-source Models				
SWE-Gym-Qwen-7B (Pan et al., 2024)	Openhands	Agent	10.6	10.0
SWE-Gym-Qwen-14B (Pan et al., 2024)	Openhands	Agent	16.4	12.7
Lingma-SWE-GPT-7B (Ma et al., 2024a)	SWE-SynInfer	Pipeline + Agent	18.2	12.0
SoRFT-Qwen-7B (Ours)	Agentless	Pipeline	**21.4**	**14.0**
32 - 72B Open-source Models				
Lingma-SWE-GPT-72B (Ma et al., 2024a)	SWE-SynInfer	Pipeline + Agent	30.2	22.0
SWE-Fixer-Qwen-72B (Xie et al., 2025a)	SWE-Fixer	Pipeline	30.2	23.3
SWE-Gym-Qwen-32B (Pan et al., 2024)	Openhands	Agent	20.6	15.3
SoRFT-Qwen-32B (Ours)	Agentless	Pipeline	**30.8**	**24.0**

React-style(Yao et al., 2022) agent-based framework, equipped with tools such as file viewing and bash command execution. The framework enables the model to autonomously invoke these tools in a React-like manner, iteratively reasoning and acting to resolve issues. They collected trajectories of Openhands invoking GPT-4o(OpenAI, 2024a) and Claude-3.5-Sonnet(Cla, 2024) for issue resolving tasks, filtered out the failed trajectories, and then fine-tuned the Qwen2.5-Coder model to serve as the SWE-Gym-Qwen model.

SWE-Fixer with SWE-Fixer-Qwen(Xie et al., 2025a). SWE-Fixer is a two-stage pipeline-based framework. First, it uses a retriever that combines BM-25 and LLMs to locate the files to be modified, and then uses LLMs to generate code edits to the files. They utilized GPT-4o(OpenAI, 2024a) to collect CoT training data, and fine-tuned the Qwen2.5-Coder model to serve as the SWE-Fixer-Qwen model.

SWE-SynInfer with Lingma-SWE-GPT(Ma et al., 2024a). SWE-SynInfer is a hybrid framework that combines pipeline design with agent-based capabilities. In the first stage, the model sequentially analyzes the repository structure, file skeletons, and code snippets to generate a detailed modification plan. Then, it provides tools such as file viewing, allowing the model to invoke these tools in a react manner and generate the final code edit. Similar to Openhands(Wang et al., 2024a), they collected trajectories of SWE-SynInfer invoking GPT-4o(OpenAI, 2024a) for issue resolving tasks, filtered out the failed trajectories, and then fine-tuned the Qwen2.5-Coder model to serve as the Lingma-SWE-GPT model.

4.5 Implementation Details

In this subsection, we will introduce the data construction details, fine-tuning details, and reinforcement learning details.

Data Construction. To construct our training dataset, we curate a collection of high-quality open-source Python projects from *seart-ghs* [3] by applying a set of stringent criteria. Specifically, we select repositories that satisfy the following conditions: (1) at least 1,000 issues, (2) at least 1,000 pull requests (PRs), (3) a minimum of 100 stars, (4) inclusion of an appropriate license, (5) exclusion of forked repositories, and (6) absence from the SWE-Bench test dataset to prevent data contamination. Through this rigorous selection process, we identify a final set of 660 repositories. From this pool, we further select 100 repositories to generate the SFT CoT data. We extract 30,000 (issue, PR) pairs from these repositories and formulate corresponding subtasks. We sample CoT data using Claude-3.5-Sonnet (Cla, 2024) and filter out instances where the final answer does not align with the subtask's ground truth, retaining 60k train-

[3] https://seart-ghs.si.usi.ch/

ing samples. We also crawl the (issue, PR) data from the remaining repositories and construct the corresponding subtasks. During the reinforcement learning phase, we randomly select 30k samples for training.

Fine-tuning. We employ FastChat (Zheng et al., 2023) framework with full sharding strategy and CPU offload strategy implemented by Pytorch FSDP [4] for our 7B model fine-tuning, and employ DeepSpeed (Rasley et al., 2020) for our 32B model fine-tuning. The training process is conducted on 4x8 96G H20 GPUs. We also utilize flash-attention-2 (Dao, 2024) to reduce memory overhead and speed up the training process. We set the global batch size to 128 and train for 2 epochs. We apply cosine learning rate decay with a maximum learning rate of 1e-5 and 3% warm-up steps.

Reinforcement Learning. We employ Open-RLHF (Hu et al., 2024) framework for our PPO (Schulman et al., 2017) implementation. The training process is conducted on 4x8 96G H20 GPUs. We also utilize ray (Moritz et al., 2018), DeepSpeed (Rasley et al., 2020), flash-attention-2 (Dao, 2024) and vllm (Kwon et al., 2023) to reduce GPU memory overhead and speed up the training process. We set temperature to 1.0 to sample completions for each prompt.

5 Results and Analysis

SoRFT achieves SOTA performance among open-source LLMs. Table 1 presents the performance of SoRFT and fine-tuned open-source LLMs on the issue-resolving task on SWE-bench Verified and SWE-bench Lite. We categorize the open-source models into two groups based on their parameter sizes: (1) 7-14B open-source LLMs, and (2) 32-72B open-source LLMs. The LLMs trained with SoRFT achieved state-of-the-art (SOTA) performance among models of the same parameter size and even slightly outperforms some larger models. On SWE-bench Verified, SoRFT-Qwen-7B outperforms SWE-Gym-Qwen-32B (21.4 vs. 20.6). SoRFT-Qwen-32B even outperforms Lingma-SWE-GPT-72B (30.8 vs. 30.2), despite the latter having significantly more parameters.

While OpenHands achieves optimal performance with proprietary models, the SWE-Gym model, specifically fine-tuned for OpenHands, underperforms compared to others. This discrepancy may arise from the challenges of constructing supervision signals for intermediate steps in agent framework, whereas a pipeline framework can establish supervision signals for different stages. This allows for finer-grained filtering of CoT data and more precise reward calculation.

[4] https://pytorch.org/docs/stable/fsdp.html

Table 2: Performance comparison of model with different training strategy on SWE-bench Verified.

Model	%Resolved	%Applied
Qwen2.5-Coder-7B-Instruct	7.6	55.6
+ SFT	18.0	85.2
+ SFT + RL (Our SoRFT-Qwen-7B)	**21.4**	**95.6**
Qwen2.5-Coder-32B-Instruct	25.6	84.4
+ SFT	28.8	90.6
+ SFT + RL (Our SoRFT-Qwen-32B)	**30.8**	**95.8**

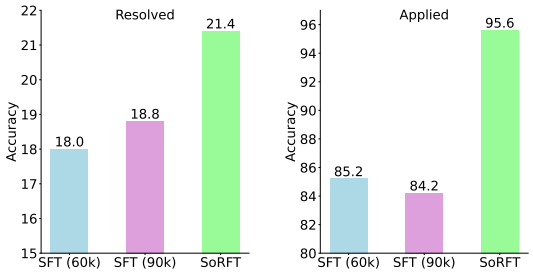

Figure 3: Performance of models trained with different training strategies.

SoRFT achieves higher accuracy than solely supervised fine-tuning. As demonstrated in Table 2, full SoRFT training consistently outperforms SFT alone for both 7B and 32B models. Both %Resolved and %Applied metrics indicate that SoRFT enhances the model's ability to resolve issues. Since the SoRFT utilized more data than SFT in the experiments of Table 2, to more fairly evaluate the effectiveness of SoRFT, we conducted an ablation on the amount of training data. As illustrated in Figure 3, even when SFT was trained on the same amount of data as SoRFT (90k samples), its performance still fell short of SoRFT's results. Increasing the training data for SFT from 60k to 90k samples slightly improved the %resolved but decreased the %Applied, indicating that excessive SFT might impair the model's generalization ability. In contrast, SoRFT was able to more robustly enhance the model's issue resolving capability.

SoRFT effectively improve performance on issue resolving subtasks. We conducted a detailed evaluation of the impact of SoRFT on various subtasks of issue resolving using the SWE-Bench Veri-

Table 3: Performance comparison on issue resolving subtasks.

Model	%File Hit	%Func Hit	%Line Hit
Qwen2.5-Coder-7B-Instruct	59.8	51.2	17.2
SoRFT-Qwen-7B	**77.8**	**66.4**	**23.6**

Table 4: Ablation results on reward rules.

Reward Rule	%Resolved	%Applied
Hit score	18.4	91.4
exact-match score	18.2	90.6
F_1 score	20.8	94.2
F_β score	**21.4**	**95.6**

fied dataset. Specifically, we measured the model's hit rates on three subtasks: file localization, function localization, and line localization. The results presented in Table 3 demonstrate that SoRFT training enhances the model's performance across these subtasks.

The robustness of reward rules is crucial for reinforcement learning. We conducted ablation study on three reward rules: hit score, exact-match score, F_1 score and F_β score. Hit score is a binary reward (1.0/0.0) based on whether the response contains ground-truth elements. Exact-match score is a binary reward (1.0/0.0) awarded only for responses that exactly match the ground truth. F_1 provides a balanced measure of response precision and recall, while F_β places more emphasis on the recall value. As shown in Table 4, training with F_β score achieves the best performance. We analyzed the training process of the three reward scores and found that: Hit scores are prone to reward hacking, LLMs just generate longer answers to hit ground-truth; Exact-match scores' sparse reward causes training plateaus around 0.2 reward, hindering further performance optimization; F_β score achieves better trade-off between precision and recall, stabilizing and accelerating convergence.

SoRFT also enhances performance on general code tasks. We also conduct evaluations on two general code tasks: LiveCodeBench (Jain et al., 2024) and RepoQA (Liu et al., 2024). LiveCodeBench focuses on self-contained code generation scenarios, while RepoQA evaluates the ability of LLMs to extract information from long-context code content. The results in Table 5 indicate that SoRFT also has the potential to enhance code-related tasks beyond issue resolving. There is a large amount of development process data in the open-source community, which remains untapped in current LLM training process. Approaches like SoRFT have the potential to utilize these data to further improving the capabilities of code LLMs.

Table 5: Performance comparison on LiveCodeBench and RepoQA.

Model	LiveCodeBench	RepoQA
Qwen2.5-Coder-7B-Instruct	34.18	85.0
SoRFT-Qwen-7B	**34.64**	**90.0**

6 Related Work

LLM Training for Issue Resolving. To enhance the issue resolving capabilities of open-source LLMs, several research works (Ma et al., 2024a; Xie et al., 2025a; Ma et al., 2024b; Pan et al., 2024) have attempted to use software development resources from the open-source community to construct training data and fine-tune open-source LLMs. Pan et al. (2024) crawled open-source repositories and utilized closed-source models (e.g., GPT-4o (OpenAI, 2024a) and Claude-3.5-Sonnet (Cla, 2024)) to generate Openhands (Wang et al., 2024a,b) Agent trajectories, and filtered them through unit tests. Then they used the trajectories to fine-tune the Qwen (Hui et al., 2024) model, enabling it to serve as the base model for Openhands. Ma et al. (2024a) used GPT-4o to generate Agent trajectories on open-source repository issues, and fine-tuned an open-source model with the filtered trajectories. Pan et al. (2024) generated CoT data for and edit generation tasks using GPT-4o, and fine-tuned an open-source model to apply it to SWE-Fixer RAG pipeline. All the above work used SFT to fine-tune models. To the best of our knowledge, we are the first work to leverage reinforced fine-tuning (Luong et al., 2024) to enhance the issue-resolving capabilities of LLMs.

Reinforcement Learning with Rule-based Reward. Since OpenAI released o1 (OpenAI, 2024b) model, many efforts have attempted to enhance LLMs' long-form reasoning capabilities through rule-based reinforcement learning. DeepSeek's R1 (Guo et al., 2025) model with rule-based GRPO (Shao et al., 2024) further demonstrates the potential of rule-based rewards. Team et al. (2025) released Kimi-k1.5, also trained with rule-based reinforcement learning. The research community (Zeng et al., 2025; Xie et al., 2025b) has also been working on replicating rule-based

reinforcement learning process. Pan et al. (2025) trained a 3B model with PPO (Schulman et al., 2017) on the Countdown task and observed "*Aha moment*" (Guo et al., 2025) phenomenon, aligns closely with the behavior of R1. Zeng et al. (2025) trained a 7B model using PPO on Math task and observed that response length initially decreased and then increased. Previous work mainly focused on mathematical tasks, where rewards can be straightforwardly computed based on ground truth. In this paper, we improve the performance of open-source models in issue-resolving framework through subtask-oriented rule-based reinforcement learning.

7 Conclusion

In this paper, we propose SoRFT, a subtask-oriented reinforced fine-tuning approach that enhances LLMs' issue-resolving capabilities. By leveraging rule-based reinforcement learning, SoRFT improves performance while ensuring better generalization. Our results demonstrate its effectiveness as a cost-efficient alternative to commercial models.

Limitations

The False Negatives of Rule-based Rewards. The correct resolution to an issue is often not unique. Relying solely on rule-based rewards by comparing the LLM's response to a single ground truth may incorrectly classify valid solutions as failures. To address this limitation, future work could incorporate unit test execution results as a more objective and fair measure of code edit quality.

Experiments were conducted only in the Python repositories. Due to the lack of a multilingual SWE-Bench test set, our experiments were limited to Python repositories. However, since SoRFT is a language-agnostic framework, we believe it has the potential to improve the issue-resolving capabilities of LLMs in other languages.

Acknowledgments

This work is supported by National Key Research and Development Program of China (Grant No. 2023YFB4503803).

References

2024. Codeqwen1.5-7b-opendevin. https://huggingface.co/OpenDevin/CodeQwen1.5-7B-OpenDevin.

2024. Introducing the next generation of claude. https://www.anthropic.com/news/claude-3-family.

Tri Dao. 2024. FlashAttention-2: Faster attention with better parallelism and work partitioning. In International Conference on Learning Representations (ICLR).

Kawin Ethayarajh, Winnie Xu, Dan Jurafsky, and Douwe Kiela. 2023. Human-centered loss functions (halos). Technical report, Contextual AI.

Kawin Ethayarajh, Winnie Xu, Niklas Muennighoff, Dan Jurafsky, and Douwe Kiela. 2024. Kto: Model alignment as prospect theoretic optimization. arXiv preprint arXiv:2402.01306.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:2401.14196.

Jian Hu, Xibin Wu, Zilin Zhu, Xianyu, Weixun Wang, Dehao Zhang, and Yu Cao. 2024. Openrlhf: An easy-to-use, scalable and high-performance rlhf framework. arXiv preprint arXiv:2405.11143.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. arXiv preprint arXiv:2409.12186.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can language models resolve real-world github issues? In The Twelfth International Conference on Learning Representations.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. 2023. Efficient memory management for large language model serving with pagedattention. In Proceedings of the 29th Symposium on Operating Systems Principles, pages 611–626.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2024. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. Advances in Neural Information Processing Systems, 36.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. Science, 378(6624):1092–1097.

Jiawei Liu, Jia Le Tian, Vijay Daita, Yuxiang Wei, Yifeng Ding, Yuhan Katherine Wang, Jun Yang, and Lingming Zhang. 2024. Repoqa: Evaluating long context code understanding. arXiv preprint arXiv:2406.06025.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. arXiv preprint arXiv:2402.19173.

Trung Quoc Luong, Xinbo Zhang, Zhanming Jie, Peng Sun, Xiaoran Jin, and Hang Li. 2024. Reft: Reasoning with reinforced fine-tuning. arXiv preprint arXiv:2401.08967.

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

Zexiong Ma, Shengnan An, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2024b. Repository structure-aware training makes slms better issue resolver. arXiv preprint arXiv:2412.19031.

Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024c. Compositional api recommendation for library-oriented code generation. In Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension, pages 87–98.

Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. 2018. Ray: A distributed framework for emerging {AI} applications. In 13th USENIX symposium on operating systems design and implementation (OSDI 18), pages 561–577.

Tong Mu, Alec Helyar, Johannes Heidecke, Joshua Achiam, Andrea Vallone, Ian Kivlichan, Molly Lin, Alex Beutel, John Schulman, and Lilian Weng. 2024. Rule based rewards for language model safety. arXiv preprint arXiv:2411.01111.

OpenAI. 2023. Gpt-4. https://openai.com/index/gpt-4-research/.

OpenAI. 2024a. Gpt-4o. https://openai.com/index/hello-gpt-4o/.

OpenAI. 2024b. o1-preview. https://openai.com/index/introducing-openai-o1-preview/.

OpenAI. 2024c. Swe-bench verified.

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

Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. 2025. Tinyzero. https://github.com/Jiayi-Pan/TinyZero. Accessed: 2025-01-24.

Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. In Proceedings of NeurIPS.

Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pages 3505–3506.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

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

Kimi Team, Angang Du, Bofei Gao, Bowei Xing, Changjiu Jiang, Cheng Chen, Cheng Li, Chenjun Xiao, Chenzhuang Du, Chonghua Liao, et al. 2025. Kimi k1. 5: Scaling reinforcement learning with llms. arXiv preprint arXiv:2501.12599.

Qwen Team. 2024a. Code with codeqwen1.5.

SWE-Bench Team. 2024b. Swe-bench lite.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024a. Executable code actions elicit better llm agents. arXiv preprint arXiv:2402.01030.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35:24824–24837.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. arXiv preprint arXiv:2308.08155.

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

Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025a. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. arXiv preprint arXiv:2501.05040.

Tian Xie, Qingnan Ren, Yuqian Hong, and Zitian Gao. 2025b. Logic-rl. https://github.com/Unakar/Logic-RL. Accessed: 2025-02-03.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. arXiv preprint arXiv:2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629.

Weihao Zeng, Yuzhen Huang, Wei Liu, Keqing He, Qian Liu, Zejun Ma, and Junxian He. 2025. 7b model and 8k examples: Emerging reasoning with reinforcement learning is both effective and efficient. https://hkust-nlp.notion.site/simplerl-reason. Notion Blog.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. 2024. Commit0: Library generation from scratch. arXiv preprint arXiv:2412.01769.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric. P Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. Preprint, arXiv:2306.05685.

This is the Appendix of the paper: *SoRFT: Issue Resolving with Subtask-oriented Reinforced Fine-Tuning*.

A Evaluation Details

All evaluation experiments are conducted on 4 96G H20 GPUs with vllm (Kwon et al., 2023) framework. We employ the official evaluation scripts [5] from SWE-Bench repository for SWE-Bench Verified and SWE-Bench Lite. On LiveCodeBench (Jain et al., 2024), we evaluate on the code_generation_lite *releave_v4* version, which contains 713 problems released between May 2023 and Sep 2024. On RepoQA (Liu et al., 2024), we evaluate on Python repositories with *similarity threshold=0.9*.

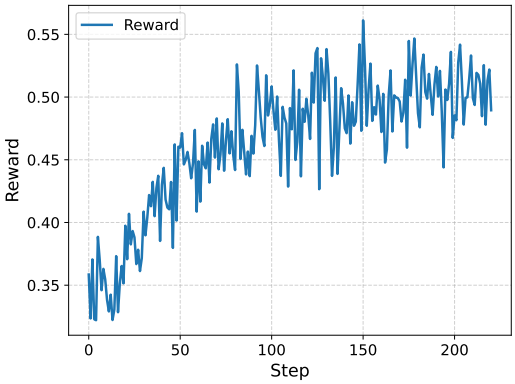

Figure 4: Reward over PPO training steps.

B PPO Training Details

To support rule-based PPO training, we implement a customized reward function to replace the original reward model scoring function in the OpenRLHF (Hu et al., 2024) framework. We set train batch size to 64, the Adam learning rate for the actor model to 5e-7, and the Adam learning rate for the critic model to 9e-6. As shown in Figure 4, using our designed reward rules, the reward steadily increases as the training steps progress.

C Additional Reasoning Example

We observed that the model tends to engage in more detailed reasoning after SoRFT training on 7B model, which bears some similarity to the "aha moment" observed by DeepSeek during the R1 training process (Guo et al., 2025). In Example 1, before concluding its reasoning, the SoRFT-Qwen-7B actively explores whether any potential answers have been overlooked and successfully identifies the ground-truth function *sql_flush*.

D Subtask Prompt

We designed data generation prompts for different subtasks based on the prompts used at corresponding stages of Agentless (Xia et al., 2024). Each subtask's prompt consists of four parts: subtask description, issue, context, and output format instructions. LLMs are required to analyze the issue description, select content from the context that aligns with the subtask description, and provide the final answer according to the specified output format. Example 2-5 is the prompt utlized to generate CoT data for each subtask.

[5] https://github.com/swe-bench/SWE-bench

Example 1: Additional Reasoning Example

Input:
Identify all locations that need inspection or editing to fix the problem.
Issue: Simplify signature of DatabaseOperations.execute_sql_flush()
...
Output:
Thought:
The issue is about simplifying the execute_sql_flush method's signature from (self, using, sql_list) to just (self, sql_list).
...
Additionally, there's a sql_flush() method (line 401) that's related to this functionality and might need to be updated as well.
...
Answer:
django/db/backends/base/operations.py
class: BaseDatabaseOperations
function: sql_flush
...

Example 2: File Localization Prompt

Please look through the following GitHub problem description and Repository structure and provide a list of files that one would need to edit to fix the problem.

GitHub Problem Description:
{issue}
Repository Structure:
{repository_structure}
Please think step by step, provide the full path and return at most 5 files.
The returned files should be separated by new lines ordered by most to least important.
For example:
```
file1.py
file2.py
```

Your reasoning should start with "### Thought:", and your answer should start with "### Answer:".

Example 3: Function Localization Prompt

Please look through the following GitHub Problem Description and the Skeleton of Relevant Files.
Identify all locations that need inspection or editing to fix the problem, including directly related areas as well as any potentially related global variables, functions, and classes.
For each location you provide, either give the name of the class, the name of a method in a class, the name of a function, or the name of a global variable.

GitHub Problem Description:
{issue}
Skeleton of Relevant Files:
{file_skeleton}
Please provide the complete set of locations as either a class name, a function name, or a variable name.
Note that if you include a class, you do not need to list its specific methods.
You can include either the entire class or don't include the class name and instead include specific methods in the class.
For example:
```
full_path1/file1.py
function: my_function_1
class: MyClass1
function: MyClass2.my_method
```

Please think step by step before returning the locations.
Your reasoning should start with "### Thought:", and your answer should start with "### Answer:".

Example 4: Line Localization Prompt

Please review the following GitHub problem description and relevant files, and provide a set of locations that need to be edited to fix the issue.
The locations can be specified as class names, function or method names, or exact line numbers that require modification.

GitHub Problem Description:
{issue}
File Contents:
{file_contents}
Please provide the class name, function or method name, or the exact line numbers that need to be edited.
For example:
```
full_path1/file1.py
line: 10
class: MyClass1
line: 51
```

Please think step by step before returning the location(s).
Your reasoning should start with "### Thought:", and your answer should start with "### Answer:".

Example 5: Code Edit Generation Prompt

You will be provided with an issue statement explaining a problem to resolve and a partial code base. Please first localize the bug based on the issue statement, and then generate **SEARCH/REPLACE** edits to fix the issue.
GitHub Problem Description:
{issue}
Code Content:
{code_content}
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*

For 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 the **SEARCH/REPLACE** edit in blocks ```python...```.

Your reasoning should start with "### Thought:", and your answer should start with "### Answer:".