SWE-FIXER: TRAINING OPEN-SOURCE LLMS FOR EFFECTIVE AND EFFICIENT GITHUB ISSUE RESOLUTION

Chengxing Xie[*1,2] **Bowen Li**[*1] **Chang Gao**[*1,3]
He Du[1] **Wai Lam**[3] **Difan Zou**[4] **Kai Chen**[1 †]
[1]Shanghai AI Laboratory [2]Xidian University [3]The Chinese University of Hong Kong
[4]The University of Hong Kong

ABSTRACT

Large Language Models (LLMs) have demonstrated remarkable proficiency across a variety of complex tasks. One significant application of LLMs is in tackling software engineering challenges, particularly in resolving real-world tasks on GitHub by fixing code based on the issues reported by the users. However, many current approaches rely on proprietary LLMs, which limits reproducibility, accessibility, and transparency. The critical components of LLMs for addressing software engineering issues and how their capabilities can be effectively enhanced remain unclear. To address these challenges, we introduce SWE-Fixer, a novel open-source framework designed to effectively and efficiently resolve GitHub issues. SWE-Fixer comprises two essential modules: a code file retrieval module and a code editing module. The retrieval module employs BM25 along with a lightweight model to achieve coarse-to-fine file retrieval. Subsequently, the code editing module utilizes the other model to generate patches for the identified files. Then, to mitigate the lack of publicly available datasets, we compile an extensive dataset that includes 110K GitHub issues along with their corresponding patches and train the two models of SWE-Fixer separately. We assess our approach on the SWE-Bench Lite and Verified benchmarks, achieving state-of-the-art performance among open-source models with scores of 24.7% and 32.8%, respectively. Additionally, our approach requires only two model calls per instance, making it significantly more efficient than existing methods. These results highlight the effectiveness of SWE-Fixer in real-world code-fixing scenarios. Our model, dataset, and code will be released.

1 INTRODUCTION

Large Language Models (LLMs) have demonstrated remarkable progress in code-related tasks, particularly excelling in code generation benchmarks such as HumanEval(Chen et al., 2021), MBPP (Austin et al., 2021) and LiveCodeBench (Jain et al., 2024). However, these benchmarks primarily focus on single-file scenarios with constrained context scope, failing to capture the complexity and interdependencies inherent in real-world software development. To bridge this gap and make LLMs more applicable to real-world development, researchers introduce SWE-Bench (Jimenez et al., 2023), a benchmark designed to assess LLMs' ability to resolve real-world GitHub issues by generating code patches and verifying their correctness through issue-specific test cases.

Current approaches to resolving the Github issues in SWE-Bench can be broadly categorized into two main paradigms: *agent* (Wang et al., 2024b; Ma et al., 2024; Albert Örwall, 2024) and *pipeline* (Xia et al., 2024; AppMap, 2024). Agent-based systems rely on LLMs dynamically determining the next action, allowing them to autonomously explore a codebase and resolve issues. In contrast, pipeline-based methods guide LLMs through a series of predefined steps, such as initially identifying the defective files and subsequently editing them to resolve the issue. Despite significant advancements in agent and pipeline-based frameworks, most existing solutions depend on powerful proprietary models such as GPT-4o and Claude-3.5-Sonnet. While these solutions are effective, they are often costly and lack transparency, preventing a deeper understanding and further improvements in the

[*]Equal Contribution. Corresponding to `xiechengxing34@gmail.com`, `libowen.ne@gmail.com`, or `chenkai@pjlab.org.cn`.
[†]Corresponding author.

problem-solving ability of LLMs. This motivates us to develop an open-source LLM to resolve the Github issues effectively and efficiently.

Along with these two paradigms, their corresponding training approaches are also largely different and face unique challenges. In particular, training open-source models in an agent-based framework presents several challenges. First, compared with frontier closed-source models such as Claude-3.5-Sonnet, existing open-source models lack the robust agent capabilities necessary for self-directed decision-making, long-term planning, and effective tool use, which may limit their performance on complex tasks. Second, constructing training data for agent-based methods often requires access to a real execution environment, which can be difficult to set up. Even with such an environment, the performance of the most advanced models is still not satisfactory to resolve real-world issues, making trajectory collection both expensive and inefficient. In contrast, while the pipeline-based approach could be less flexible than the agent-based approach, it simplifies data construction and model training by explicitly defining each subtask. Notably, the Agentless framework (Xia et al., 2024) employs a complex, multi-stage design that organizes problem-solving procedures into a pipeline. This approach has demonstrated competitive performance on the GitHub issue resolution task using proprietary models. For instance, the framework identifies problematic code through a multi-step process: LLMs first filter candidate files, then apply both LLM-based and dense retrieval on the selected subset, followed by LLM-guided localization of relevant classes, functions, and code lines. While this pipeline provides a structured training framework, the primary bottleneck lies in the scarcity of high-quality real-world training data for all intermediate tasks, which hinders the practical training of open-source models capable of accurately executing each step in the pipeline.

To address these challenges, we introduce SWE-Fixer, a simple yet effective pipeline-based approach for training open-source models to resolve Github issues. Unlike Agentless (Xia et al., 2024), which employs a complex pipeline, SWE-Fixer streamlines the process by reducing the number of reasoning steps, facilitating more effective model training, and significantly lowering inference costs. Our method decomposes the task into two core subtasks: code file retrieval and code editing (see Figure 1). For the retrieval task, we use a *coarse-to-fine* strategy combining BM25 for initial file retrieval and a model to identify the defective files from the BM25 results. Once the defective files are identified, the editing model generates a code patch to resolve the issue, trained using chain-of-thought data. The streamlined pipeline design facilitates easier training data construction and efficient inference, eliminating the need for complex manipulations. We curate a large-scale dataset comprising 110K instances for the training of both retrieval and editing tasks. To ensure data quality, we apply rigorous filtering techniques, making the training set both extensive and reliable. For model implementation, we fine-tune the Qwen2.5 (Qwen et al., 2024) base series models, which include a 7B retriever and a 72B editor.

SWE-Fixer achieves SOTA performance among open-source models. We evaluate SWE-Fixer on both SWE-Bench Lite and Verified, achieving 24.7% and 32.8% Best@1 performance, respectively, which are the highest among open-source model-based approaches. Furthermore, SWE-Fixer requires *only two model calls per instance*, making it significantly more efficient than existing open-source frameworks while maintaining superior performance. Notably, compared to frameworks built on proprietary models, SWE-Fixer outperforms several methods on both benchmarks, particularly those based on GPT-4, GPT-4o, and Claude-3-Opus, demonstrating exceptional efficiency and effectiveness. We also conduct a comprehensive study on training configurations for both tasks, examining the impact of data format settings on performance, the scaling trends of the editing task across different base models, to provide insights into optimizing future systems. Additionally, the trained retriever and editor from SWE-Fixer have the potential to serve as modular components in agent-based systems, further enhancing the efficiency and effectiveness of existing agent-based approaches. Our contributions can be summarized as follows:

- **State-of-the-art performance:** We propose a novel pipeline-based method that leverages open-source models, achieving state-of-the-art Best@1 performance on SWE-Bench Lite and Verified compared to other open-source model-based approaches.
- **Large-scale dataset curation:** We curate a large-scale dataset comprising 110K instances with rigorous filtering techniques to ensure both the extensiveness and reliability of the training data.
- **Comprehensive Analysis:** We provide an in-depth analysis of the data configurations for each subtask, enabling optimized task performance.

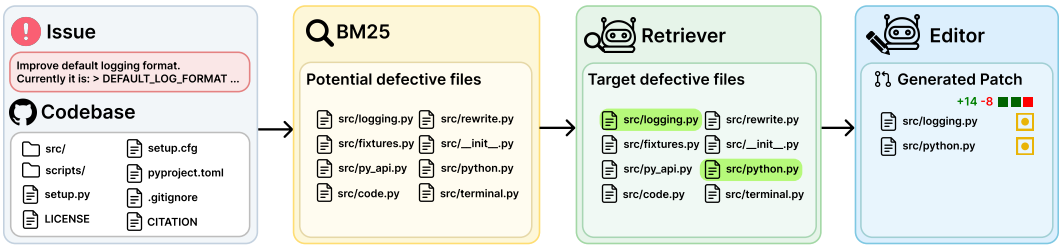


Figure 1: The pipeline of SWE-Fixer. Our framework contains two components: code file retrieval and code editing. The two frames of BM25 and Retriever indicate our *coarse-to-fine* retrieval method. The editor frame represents the code editing task.

2 Related Works

Code Large Language Models The use of large language models (LLMs) for coding tasks has garnered substantial attention recently. Numerous LLMs, trained on extensive datasets of code, have been introduced, including closed-source models such as GPT-4 (Achiam et al., 2023), Gemini (Team et al., 2023), and Claude-3.5 (Anthropic, 2024) and open-source models such as Mixtral (Jiang et al., 2024), Deepseek-Coder Zhu et al. (2024), and Qwen-Coder (Hui et al., 2024). Beyond these general-purpose code models, significant efforts have been made to tackle specific software engineering challenges with specialized LLMs. For instance, several studies have introduced strategies to improve program generation (Wang et al., 2024a; Huang et al., 2023; Zheng et al., 2023; Chen et al., 2024; Liu et al., 2024a) or enhance program repair (Lin et al., 2024; Jiang et al., 2023; Xia et al., 2022). In the domain of code retrieval, researchers have developed advanced techniques for refining code representations (Bui et al., 2021; Li et al., 2024; Martinez-Gil, 2024; Saieva et al., 2023) and improving fault retrieval (Qin et al., 2024; Du et al., 2024).

LLMs for GitHub Issue Solving In software engineering, resolving repository issues with LLMs has gained significant attention. A widely recognized benchmark for this task is SWE-Bench (Jimenez et al., 2023), which consists of 2,294 issues curated from 12 real-world, high-quality GitHub repositories. Various agent-based approaches, including SWE-agent (Yang et al., 2024), Autocoderover (Zhang et al., 2024), OpenHands (Wang et al., 2024b), and Moatless Tools (Albert Örwall, 2024), built on proprietary models, have been proposed to enhance the performance of LLMs on SWE-Bench. These methods equip LLMs with various tools, allowing them to autonomously explore the codebase and resolve issues. SWE-SynInfer (Ma et al., 2024) and SWE-Gym (Pan et al., 2024) attempt to train open-source models in an agent-based framework to tackle these challenges. Additionally, pipeline-based methods, such as Agentless (Xia et al., 2024), have been introduced for this task, showing promising results compared to SOTA agent-based methods.

Agentless shares a similar philosophy with SWE-Fixer but differs significantly in its approach. Overall, Agentless features a highly sophisticated architectural design that leverages powerful proprietary models, making it challenging to adapt for training open-source models. In contrast, SWE-Fixer adopts a more streamlined approach. Instead of Agentless's multi-step, fine-grained retrieval process, which has proven ineffective for open-source models (see Section 5.4), SWE-Fixer employs a simple yet effective two-step retrieval strategy. Similarly, while Agentless introduces extensive complexity in the editing stage, requiring the LLM to handle patch generation, test reproduction, and file selection, SWE-Fixer focuses solely on patch generation using retrieved file content. Ultimately, SWE-Fixer offers a simpler and more robust design, facilitating easier training data construction, making it better suited for finetuning open-source models, and delivering strong results on SWE-Bench.

3 SWE-Fixer

3.1 Overview

The SWE-Fixer framework is designed to efficiently address real-world GitHub issues by generating code patches that specify the necessary modifications to a repository's codebase. We divide this task into two subtasks: code file retrieval and code editing. Identifying the correct files for modification

is a major challenge, as repositories typically contain a vast number of files. To tackle this, we introduce a coarse-to-fine retrieval approach that efficiently narrows down the search space and accurately locates the files requiring edits. Once the files are identified, a specialized editor model is employed to produce high-quality code patches that effectively resolve the reported issues. As shown in Figure 1, we adopt a structured pipeline approach to systematically optimize open-source models for each subtask. The following sections provide a detailed breakdown of these components within the SWE-Fixer framework.

3.2 Code File Retrieval

To efficiently identify relevant files for modification in a repository, we adopt a *coarse-to-fine* strategy. First, as seen in the BM25 part of Figure 1, we use BM25 (Robertson et al., 2009) to retrieve the 30 most relevant files to the issue, treating the issue description as a query and the code files as documents. Building on these initial retrieval results, we then finetune a retriever model to further refine the selection and accurately identify which files require modification. We chose BM25 over the dense retrieval method used in the Agentless (Xia et al., 2024) and Moatless Tools (Albert Örwall, 2024) because it provides a lightweight, scalable and robust approach for initial file retrieval, especially when the number of files in a repository is large. BM25 provides an efficient and effective initial retrieval mechanism, which is then complemented by a language model refinement step to narrow down the selection to the most relevant files.

Given the large potential context of all files, we take inspiration from the Agentless framework and use file documentations (skeletons) as input. A file documentation includes module docstrings, class headers, class methods, and function signatures, retaining only the signatures of class methods and the first and last five lines of functions (see example in Appendix E). This approach significantly reduces context size while preserving the essential information for effective file-level retrieval.

3.3 Code Editing

The code editing task involves generating a patch to resolve an issue based on the relevant files. Previous works on model training primarily adopt agentic approaches without explicitly treating code editing as a standalone sub-task. In this work, we identify code editing as the main bottleneck in the entire workflow and explicitly focus on improving its effectiveness. During training, we utilize gold defective files, i.e., those within the patches. For inference, we use retrieved files. To provide sufficient context, we include the content of all retrieved files, even though only a small portion of the content in these files would be modified. Including additional context helps the model better understand the issue and apply the necessary changes effectively.

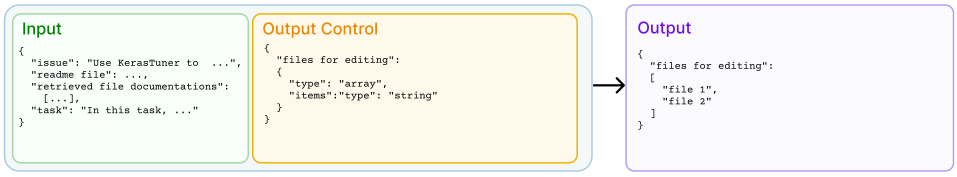
To assist the model in generating valid code modifications, we define a structured output that includes three key components (see Appendix F). The first component is the file path, which specifies the location of the file to be modified. The second is the original code block, which includes the specific code snippet to be edited along with its line numbers. The third component is the modified code block, which represents the final modification result but excludes line numbers as the new line numbers are difficult for the model to calculate. The input includes complete file content with line numbers. This approach ensures that the model can efficiently identify the lines to edit and produce valid modifications. In contrast, the standard patch format[1] requires line number calculations in the output hunk, which adds complexity. Our structured output simplifies model training and can be transformed to patches automatically for evaluation.

4 Model Training

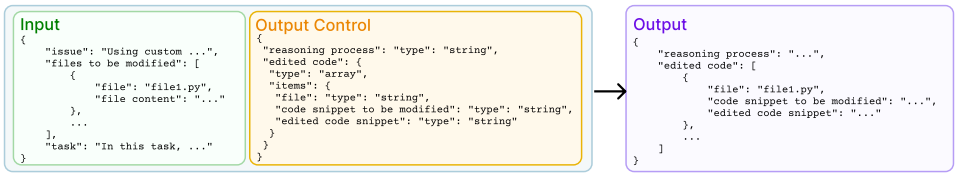
4.1 Structured Instruction Tuning

We adopt a structured approach, JsonTuning (Gao et al., 2023), to train our models, enhancing the overall pipeline performance. As shown in Figure 2, the input JSON structure includes task input elements, task instructions, and output control information, while the output JSON structure consists of task output elements. For the code file retrieval task, input elements include issues

[1] https://git-scm.com/docs/diff-generate-patch



(a) The code file retrieval task.



(b) The code editing task.

Figure 2: Structured representation of the retrieval/editing task's inputs and outputs.

and file documentations, and output elements include files for editing. For the code editing task, input elements include issues and file content, and output elements include reasoning processes, original code snippets, and modified code snippets. During training, we provide the model with a task-specific input in JSON format and expect it to generate a corresponding JSON-formatted output. The employment of JsonTuning offers several advantages: (1) By incorporating explicit task structures into the input and output representations during instruction tuning, JsonTuning enhances the model's comprehension of critical task elements and their interrelations, thereby improving generalization capabilities. (2) Given the inherently structured nature of code, JsonTuning efficiently leverages this structured information. (3) JsonTuning provides a structured output that is more robust compared to generating patches, which often involve stricter syntax and pose greater challenges for model interpretation.

Post-processing To ensure that the model-generated outputs meet the task requirements, we implement a post-processing procedure with a unified resampling strategy for both the retrieval and code editing tasks to ensure output validity. The sampling temperature is initially set to 0 for deterministic outputs. If the output fails or is invalid, the temperature is adjusted to 0.7 for further attempts. To prevent endless retries, the maximum number of attempts is set to 5. For the retrieval task, outputs are checked for JSON syntax validation, which means, triggering resampling if invalid. For the code editing task, we similarly require the model-generated results to initially pass JSON validation. Once validated, the model-generated modifications are applied to the original file. If the original code snippet cannot be located, the result is deemed invalid. Additionally, the modification is considered invalid if the modified code fails to pass syntax checks after replacement or if the corresponding test files already exist in the repository. This post-processing procedure effectively improves the correctness of the model-generated outputs in terms of format and content, ensuring that they better meet the requirements of real-world applications.

4.2 Chain-of-Thought Data Construction

For tasks that require strong reasoning capabilities, such as math or coding, LLMs can improve their performance by using Chain-of-Thought. However, the data collected from real-world scenarios typically includes only the specific codebase associated with an issue and the corresponding gold patches, without capturing the intermediate reasoning behind these patches. To address this gap, we construct Chain-of-Thought (CoT) data for the code editing task. A straightforward approach to generating CoT data involves prompting a teacher model to produce the reasoning process and then performing rejection sampling. However, this approach presents several challenges. Due to the complexity of the task, even advanced proprietary models struggle to achieve high accuracy when performing code editing independently. Additionally, without access to a real code execution environment, we are unable to verify the patches generated by the teacher model through execution, making standard rejection sampling infeasible.

To address these challenges, we adopt a methodology inspired by the *rationalization* approach in Zelikman et al. (2022). Specifically, we use the gold patches as part of the input to guide the teacher

Table 1: Performance of various models/methods on SWE-Bench Lite and Verified. ◊: Finetuned open-source models. *: All results are reported as Pass@1 or Best@1 in this table, except for SWE-Gym, which uses a specially trained 32B verifier.

Method	Model	Type	Verified	Lite
Open-source Methods w/ Proprietary Models				
RAG (Jimenez et al., 2023)	GPT-4	Pipeline	2.8	2.7
RAG (Jimenez et al., 2023)	Claude-3-Opus	Pipeline	7.0	4.3
SWE-agent (Yang et al., 2024)	Claude-3-Opus	Agent	18.2	11.7
SWE-agent (Yang et al., 2024)	GPT-4o	Agent	23.0	18.3
AppMap Navie (AppMap, 2024)	GPT-4o	Pipeline	26.2	21.7
AutoCodeRover (Zhang et al., 2024)	GPT-4	Agent	-	19.0
SWE-agent + RepoGraph (Ouyang et al., 2024)	GPT-4o	Agent	-	20.3
AutoCodeRover + RepoGraph (Ouyang et al., 2024)	GPT-4	Agent	-	21.3
OpenHands (Wang et al., 2024b)	GPT-4o	Agent	-	22.0
AutoCodeRover (Zhang et al., 2024)	GPT-4o	Agent	28.8	22.0
SWE-SynInfer (Ma et al., 2024)	GPT-4o	Agent	31.8	20.7
SWE-agent (Yang et al., 2024)	Claude-3.5-Sonnet	Agent	33.6	23.0
Agentless (Xia et al., 2024)	GPT-4o	Pipeline	38.8	32.0
Moatless Tools (Albert Örwall, 2024)	Claude-3.5-Sonnet-`20241022`	Agent	-	38.3
AutoCodeRover-v2.0 (Zhang et al., 2024)	Claude-3.5-Sonnet-`20241022`	Agent	-	46.2
Agentless (Xia et al., 2024)	Claude-3.5-Sonnet-`20241022`	Pipeline	50.8	40.7
OpenHands (Wang et al., 2024b)	Claude-3.5-Sonnet-`20241022`	Agent	53.0	41.7
Open-source Methods w/ Open-source Models				
RAG (Jimenez et al., 2023)	SWE-Llama-13B◊	Pipeline	1.2	1.0
AutoCodeRover (Liu et al., 2024b)	Qwen2-72B-Instruct	Agent	-	9.3
SWE-Gym (Best@1) (Pan et al., 2024)	SWE-Gym-32B◊	Agent	20.6	15.3
SWE-SynInfer (Ma et al., 2024)	Lingma-SWE-GPT-72B◊	Agent	30.2	22.0
SWE-Gym (Best@8 w/ Verifier)* (Pan et al., 2024)	SWE-Gym-32B◊	Agent	29.8	**26.0**
SWE-Search (Antoniades et al., 2024)	Qwen2.5-72b-Instruct	Agent	-	24.7
SWE-Fixer (Ours)	SWE-Fixer-72B◊	Pipeline	**32.8**	24.7

model in generating both the reasoning process and the corresponding patches. The model is tasked with producing a reasoning chain and code patch as if it were *unaware* of the correct answer. Detailed prompts can be found in Appendix D. We use GPT-4o for the generation, and the resulting reasoning chains are generally coherent and sound.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

We finetune the Qwen2.5 (Qwen et al., 2024) base series models to implement SWE-Fixer, which consists of a 7B code retriever and a 72B code editor. The training is conducted on 96 Nvidia A800 GPUs using the xtuner-lite framework (Contributors, 2023), with a global batch size of 96 and a 64K context window. We evaluate SWE-Fixer on two datasets: SWE-Bench Lite and SWE-Bench Verified. SWE-Bench Lite, part of the official SWE-Bench benchmark (Jimenez et al., 2023), consists of 300 selected instances optimized for efficient evaluation. SWE-Bench Verified, proposed by OpenAI, is a human-validated subset that offers a more reliable evaluation. Each instance includes a real-world GitHub issue description paired with its corresponding codebase. Model-generated code patches are assessed using developer-written unit tests, and accuracy is calculated as the percentage of instances successfully resolved.

5.2 DATASET PREPARATION

For the code file retrieval task, we filter out those where the gold defective files do not appear in the top 30 retrieved files, or where the total length exceeds the 64K context window limit. This results in 80K valid training instances for the retrieval task, denoted as `SWE-Fixer-Retrieval-80K`.

For the code editing task, due to computational resource constraints and API limits, we sample a subset of 70K instances, referred to as `SWE-Fixer-Editing-70K`, and construct CoT data for this subset, `SWE-Fixer-Editing-70K-CoT`. To explore optimal data formats and training strategies, we further sample 200 repositories, each contributing 50 instances (10K instances total),

Table 2: Performance comparison of different methods with open-source models on SWE-Bench Verified and Lite. '#Model Calls' represents the number of model calls required per instance. †: The minimum number of model calls needed. ‡: The average number of model calls needed. *: See Appendix C for the detailed calculation.

Method	Model	Verified	Lite	#Model Calls
RAG (Jimenez et al., 2023)	SWE-Llama-13B	1.2	1.0	1
AutoCodeRover (Liu et al., 2024b)	Qwen2-72B-Instruct	-	9.3	4†
SWE-Gym (Best@1) (Pan et al., 2024)	SWE-Gym-32B	20.6	15.3	29.3‡
SWE-SynInfer (Ma et al., 2024)	Lingma-SWE-GPT-72B	30.2	22.0	6†
SWE-Search (Antoniades et al., 2024)	Qwen2.5-72b-Instruct	-	24.7	200*
SWE-Fixer (Ours)	SWE-Fixer-72B	**32.8**	**24.7**	2

to create a smaller dataset, `SWE-Fixer-Train-10K`, for the ablation study (see Section 5.4). The detailed training data collection process is provided in Appendix B.

5.3 Main Results

Table 1 presents the primary results on SWE-Bench Lite and SWE-Bench Verified. We categorize the results into two groups: the first uses open-source frameworks based on proprietary models, while the second involves approaches built entirely on open-source frameworks and models.[2]

We begin by comparing SWE-Fixer with open-source frameworks based on proprietary models. Remarkably, our approach outperforms several methods across both SWE-Bench Lite and Verified, especially those based on GPT-4, GPT-4o, and Claude-3-Opus. The exception is only Agentless (Xia et al., 2024) using GPT-4o. This is encouraging, as these baselines typically involve complex agent frameworks, specifically designed for the SWE-Bench task, and rely on advanced proprietary models. Our results suggest that SWE-Fixer offers a promising, cost-effective alternative to proprietary model-based frameworks. We also note that Claude-3.5-Sonnet excels in coding tasks, delivering consistent improvements over all open-source methods, where our approach still lags behind such systems.

Among the approaches based on open-source models, our SWE-Fixer framework achieves the highest Best@1 performance. While SWE-Gym-32B (Pan et al., 2024) attains a higher Best@8 score on SWE-Bench Lite with its specially trained 32B verifier, it still falls behind SWE-Fixer on SWE-Bench Verified. SWE-Gym represents a promising direction by leveraging reinforcement learning in executable environments. However, its heavy reliance on human effort for environment construction limits its scalability. SWE-Fixer outperforms SWE-SynInfer, an agent-based method using fine-tuned models, on both SWE-Bench Lite and Verified.

Beyond effectiveness, our method also demonstrates exceptional efficiency. As shown in Table 2, apart from the RAG method, which performs poorly, SWE-Fixer requires only two model calls (7B retrieval + 72B editing) per instance, making it significantly more efficient than other methods while maintaining state-of-the-art performance. Notably, SWE-Search requires at least 200 model calls per instance yet achieves the same score as SWE-Fixer on SWE-Bench Lite, making it prohibitively expensive in comparison.

5.4 Ablation Study

5.4.1 Code File Retrieval

As shown in Table 3, we conduct a series of ablation experiments on the code file retrieval task to evaluate the impact of various input configurations on model performance. The default setup employs a 64K context window that includes the readme file along with BM25-retrieved file documentation, limited to a maximum of 30 files. The model is trained to predict a list of files requiring modification.

Irrelevant information adversely affects performance. Providing a relevant input context is critical in the code file retrieval task. Including unnecessary details, such as the entire file content, leads to a decline in performance since the retrieval task does not require such fine-grained code information. Moreover, increasing the number of input files to fully occupy the context window improves recall but reduces precision, which offers no performance gains for the overall pipeline, as shown in Table 5,

[2]Closed-source methods are omitted from Table 1, as the lack of technical transparency limits the discussion of meaningful insights.

Table 3: Ablation study of the code file retrieval task on SWE-Bench Lite. The table compares precision and recall across different methods and training datasets, using Qwen2.5-7B as the base model. *: The default setting utilizes a 64K context window, which includes the readme file and BM25-retrieved file documentation limited to 30 files. †: Fills the entire 64K context window with file documentation, which may exceed the default limit of 30 files. ‡: Trains with additional 100K editing data without CoT, which is sampled from `SWE-Fixer-Train-110K`.

Method	Training Dataset	Precision(%)	Recall(%)
BM25 Baseline			
BM25 Top-3	-	18.9	56.7
BM25 Top-30	-	2.9	86.7
Training on Limited Data			
Default setting*	**`SWE-Fixer-Train-10K`**	65.4	67.3
- Remove readme	`SWE-Fixer-Train-10K`	65.2 (↓ 0.2)	66.7 (↓ 0.6)
- Remove 30 files limit†	`SWE-Fixer-Train-10K`	64.9 (↓ 0.5)	68.7 (↑ 1.4)
- 32K context limit	`SWE-Fixer-Train-10K`	64.3 (↓ 1.1)	64.7 (↓ 2.6)
- Add file content	`SWE-Fixer-Train-10K`	59.7 (↓ 5.7)	60.3 (↓ 7.0)
Training on Full Data			
Default setting*	`SWE-Fixer-Retrieval-80K`	68.5 (↑ 3.1)	69.0 (↑ 1.7)
Mix training‡	`SWE-Fixer-Retrieval-80K` + 100K Editing Data	**69.4** (↑ 4.0)	**69.7** (↑ 2.4)

while also demanding more computational resources. Conversely, incorporating relevant information, such as the readme file, improves model performance.

Smaller context windows reduce effectiveness. A smaller context window (32K) restricts the number of input files, which may lead to the exclusion of target defective files. This limitation significantly lowers recall within the input context, ultimately resulting in poorer overall performance compared to a model using a larger 64K context window.

Larger datasets improve model performance. Expanding the training dataset size from 10K to 80K significantly boosts the model's performance. Additionally, incorporating data from both the retrieval and editing tasks during training further enhances retrieval performance. This improvement highlights the beneficial influence of editing task data on the retrieval task.

5.4.2 Code Editing

Table 4: Ablation study of the code editing task on SWE-Bench Lite with gold defective code files as input using Qwen2.5-72B and the `SWE-Fixer-Train-10K`. *Cls&Func* refers to classes and functions. *: The default setting uses only file content with line numbers. †: Contains only Cls&Func content, without file content.

Method	Resolve (%)
Default setting*	20.0
- Only Cls&Func Content†	18.0 (↓ 2.0)
- Add Cls&Func Name	**22.0** (↑ 2.0)
- Add readme	19.0 (↓ 1.0)
- Remove Line Number	14.0 (↓ 6.0)

Table 5: Impact of different retrieval training methods of the 7B model on the overall pipeline performance of SWE-Bench Lite, using the 72B editor model. Both the retrieval and editor models are trained on `SWE-Fixer-Train-10K`. *: Fills the entire 64K context window, which may exceed the default limit of 30 files. †: In this setup, the retriever is trained to additionally retrieve class and function names, where the 72B editor model is also specifically trained to incorporate the input change.

Method	Pre/Recall(%)	Resolve(%)
Default setting	**65.4**/67.3	**16.3**
- Remove 30 file limit*	64.9/**68.7**	16.0 (↓ 0.3)
- Also retrieve Cls&Func†	54.0/56.0	15.7 (↓ 0.6)

As shown in Table 4, we also conduct a detailed ablation study on the code editing task to evaluate the impact of different data configurations. All experiments use gold input (oracle defective files). The default configuration includes complete file content with line numbers, providing better contextual understanding and precise localization.

Enhanced location information improves performance. Including line numbers acts as an anchor, helping LLMs locate and edit specific code snippets more effectively, which improves performance. Adding oracle class and function names that require editing also enhances performance by providing additional location-specific details. However, this additional information increases the complexity of retrieval tasks and reduces the overall pipeline performance given retrieved class and function names (see Table 5).

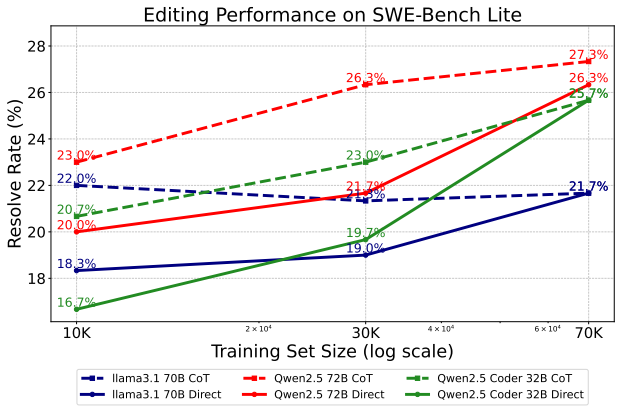


Figure 3: Performance of the editing task on SWE-Bench Lite across various models and training settings with gold defective code files as input. The x-axis is in logarithmic scale. Solid lines indicate methods that generate modified code directly, while dashed lines represent methods that incorporate a reasoning step before generating the modified code.

Redundant or insufficient information reduces performance. The readme file, while useful in retrieval tasks, introduces high-level abstract information that is irrelevant to the editing task. Editing tasks require detailed understanding of flawed code, and the inclusion of readme information detracts from this, resulting in lower performance. Insufficient information, such as input limited to class and function content, also negatively impacts performance. This suggests that class and function-only inputs omit essential context needed for effective editing.

5.5 Scaling Trends of the Code Editing Task

Compared to the retrieval task, LLMs perform significantly worse on the editing task, making it the primary bottleneck in overall task performance. To further investigate LLMs' capabilities in code editing, we analyze their performance by training on datasets of varying scales and using gold input to test their editing ability on SWE-Bench Lite, as shown in Figure 3. The direct training method demonstrates consistent performance improvements across models as the training data size increases. Stronger models exhibit steeper scaling trends compared to weaker models. For instance, while Qwen2.5-Coder-32B initially lags behind Llama3.1-70B with 10K training instances, it gradually surpasses Llama3.1-70B as more training data is available. Notably, even at 70K instances, the performance curves for these models have yet to plateau, suggesting that further increasing the training data size could yield additional performance gains for direct training. In contrast, CoT training exhibits varying scaling trends across models. For Qwen2.5-Coder-32B and Qwen2.5-72B, the CoT method demonstrates a consistent upward trend in performance, significantly outperforming direct training. However, for Llama-3.1-70B, which may lack sufficient capacity in this domain, increasing the CoT training data does not lead to sustained performance improvements. Additionally, as the training dataset grows, the performance gap between CoT and direct training narrows.

6 Conclusion

In this paper, we present SWE-Fixer, an open-source framework designed to efficiently and effectively address real-world software engineering challenges using finetuned open-source models. SWE-Fixer adopts a pipeline-based approach, dividing the task into two subtasks: code file retrieval and code editing, requiring only two steps to generate the final results. To support model training, we curate a large-scale, real-world dataset and construct task-specific training data for both subtasks. Our method demonstrates impressive performance on SWE-Bench Lite and Verified, achieving the highest Best@1 performance among open-source model-based approaches, while maintaining low inference steps and minimal computational overhead. Notably, SWE-Fixer outperforms several methods based on proprietary models, including those leveraging GPT-4, GPT-4o, and Claude-3-Opus. By offering a simple yet effective approach to training models for software engineering tasks, SWE-Fixer lowers barriers for the community and fosters further innovation in this domain.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

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

Anthropic. Claude 3.5 sonnet. `https://www.anthropic.com/news/claude-3-5-sonnet`, 2024.

Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285*, 2024.

AppMap. Appmap navie. `https://appmap.io/blog/2024/06/20/appmap-navie-swe-bench-leader/`, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 511–521, 2021.

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.

Mouxiang Chen, Hao Tian, Zhongxin Liu, Xiaoxue Ren, and Jianling Sun. Jumpcoder: Go beyond autoregressive coder via online modification. *arXiv preprint arXiv:2401.07870*, 2024.

XTuner Contributors. Xtuner: A toolkit for efficiently fine-tuning llm. `https://github.com/InternLM/xtuner`, 2023.

Xiaohu Du, Ming Wen, Jiahao Zhu, Zifan Xie, Bin Ji, Huijun Liu, Xuanhua Shi, and Hai Jin. Generalization-enhanced code vulnerability detection via multi-task instruction fine-tuning. *arXiv preprint arXiv:2406.03718*, 2024.

Chang Gao, Wenxuan Zhang, Guizhen Chen, and Wai Lam. Jsontuning: Towards generalizable, robust, and controllable instruction tuning. *arXiv preprint arXiv:2310.02953*, 2023.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*, 2023.

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

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

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1430–1442. IEEE, 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? *arXiv preprint arXiv:2310.06770*, 2023.

Haochen Li, Xin Zhou, and Zhiqi Shen. Rewriting the code: A simple method for large language model augmented code search. *arXiv preprint arXiv:2401.04514*, 2024.

Bo Lin, Shangwen Wang, Ming Wen, Liqian Chen, and Xiaoguang Mao. One size does not fit all: Multi-granularity patch generation for better automated program repair. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1554–1566, 2024.

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. *Advances in Neural Information Processing Systems*, 36, 2024a.

Xiangyan Liu, Bo Lan, Zhiyuan Hu, Yang Liu, Zhicheng Zhang, Fei Wang, Michael Shieh, and Wenmeng Zhou. Codexgraph: Bridging large language models and code repositories via code graph databases. *arXiv preprint arXiv:2408.03910*, 2024b.

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

Jorge Martinez-Gil. Improving source code similarity detection through graphcodebert and integration of additional features. *arXiv preprint arXiv:2408.08903*, 2024.

Siru Ouyang, Wenhao Yu, Kaixin Ma, Zilin Xiao, Zhihan Zhang, Mengzhao Jia, Jiawei Han, Hongming Zhang, and Dong Yu. Repograph: Enhancing ai software engineering with repository-level code graph. *arXiv preprint arXiv:2410.14684*, 2024.

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.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*, 2024.

Qwen, :, 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, 2024. URL `https://arxiv.org/abs/2412.15115`.

Stephen Robertson, Hugo Zaragoza, et al. The probabilistic relevance framework: Bm25 and beyond. *Foundations and Trends® in Information Retrieval*, 3(4):333–389, 2009.

Anthony Saieva, Saikat Chakraborty, and Gail Kaiser. Reinforest: Reinforcing semantic code similarity for cross-lingual code search models. *arXiv preprint arXiv:2305.03843*, 2023.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024a.

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

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Practical program repair in the era of large pre-trained language models. *arXiv preprint arXiv:2210.14179*, 2022.

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

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

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah Goodman. Star: Bootstrapping reasoning with reasoning. *Advances in Neural Information Processing Systems*, 35:15476–15488, 2022.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Lin Zheng, Jianbo Yuan, Zhi Zhang, Hongxia Yang, and Lingpeng Kong. Self-infilling code generation. In *Forty-first International Conference on Machine Learning*, 2023.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.

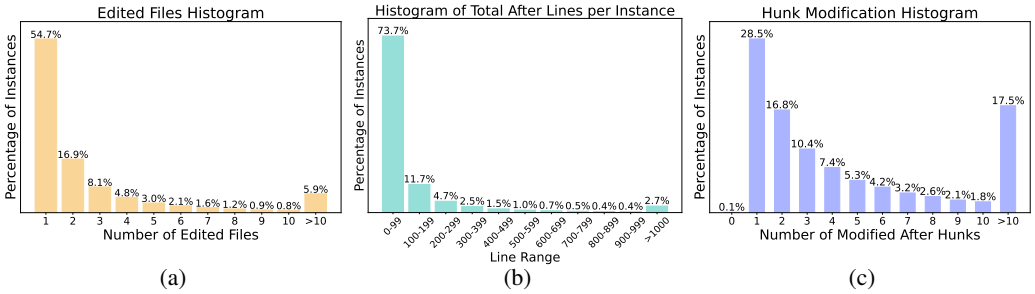
A DETAIL COMPARISON WITH AGENTLESS

Agentless shares a similar philosophy with our SWE-Fixer but differs significantly in its approach. Overall, Agentless features a highly sophisticated architectural design that leverages powerful proprietary models, this complexity poses challenges for training open-source models. In the retrieval stage, Agentless employs a multi-step process, including LLM-prompted file retrieval, LLM-filtered dense retrieval, and multi-level fine-grained retrieval, ranging from class and function levels to individual lines. However, attempts to replicate such intricate cascaded steps has proven ineffective for open-source model training (see Section 5.4). To address this, SWE-Fixer adopts a simpler yet effective two-step retrieval approach, combining BM25 for coarse retrieval with a trained retrieval model to further identify defective files. This streamlined method is not only cost-efficient but also demonstrates strong performance in our experiments. Similarly, Agentless introduces significant complexity in the editing stage by requiring the LLM to generate patches, reproduce tests, and select test files, resulting in a heavy and demanding pipeline that is difficult for open-source LLMs to learn and execute. In contrast, SWE-Fixer simplifies the editing task by generating patches using only retrieved file content and incorporating a chain-of-thought (CoT) mechanism to enhance performance. Ultimately, compared to Agentless, SWE-Fixer offers a simpler and more robust design that facilitates easier training data construction, is better suited for finetuning open-source models, and delivers impressive results on SWE-Bench.

B TRAINING DATA COLLECTION

In line with the data collection methodology outlined in SWE-Bench , we gather high-quality issues, pull requests, and codebases from GitHub repositories for training purposes. Subsequently, we apply a filtering process to enhance data quality and eliminate overly complex training examples.

Repository Collection We use Github REST API to crawl a list of initial repositories. We select Python repositories that have more than 100 pull requests (PRs). In the official SWE-Bench code base, an instance extraction script is used to crawl SWE-Bench-style instances, each corresponding to an issue with a PR and a set of potential unit tests. We find that this script relies on string match [3] to identify valid issue PR pairs. Therefore, it has a large chance to miss out many instances. We instead rely on GitHub events[4] to conduct the crawling job. Eventually, for the raw data, we collect 2.3K repositories and 331K instances, where repositories within SWE-Bench have been excluded.



(a) (b) (c)

Figure 4: We sample a subset of the crawled raw data to analyze the statistics on real-world data. (a) The histogram of the number of edited files. (b) The distribution of the number of edited code lines. (c) The histogram shows the number of edited code hunks.

Data Statistics As shown in Figure 4a, most real-world instances involve a small number of edited files. Specifically, 54.7% of instances involve modifications to only one file, and nearly 80% involve no more than three files. Figure 4b shows the distribution of modified lines, with 73.7% of instances

[3] https://docs.github.com/en/issues/tracking-your-work-with-issues/using-issues/linking-a-pull-request-to-an-issue#linking-a-pull-request-to-an-issue-using-a-keyword

[4] https://docs.github.com/en/rest/using-the-rest-api/issue-event-types?apiVersion=2022-11-28

involving modifications to 0–99 lines and over 85% involving up to 200 lines. Similarly, Figure 4c reveals that instances involving more modification regions (hunks) occur less frequently.

Data Filtering To address the challenges posed by the messy nature of real-world data, we apply instance-level data filtering to ensure higher data quality. For the sake of efficiency, we sample a subset of 140K instances, parse the oracle code patches and discard instances whose patches cannot be parsed. Then we filter out instances where the number of edited files (excluding test files) is more than three. This decision is based on our observation in the data statistics and editing more than three files introduces excessive complexity, hindering effective model training. This process results in a train set of 110K instances, `SWE-Fixer-Train-110K`.

C MODEL CALLS CALCULATION FOR SWE-SEACH

Since SWE-Search has not released its execution trajectory, we can estimate the required inference steps based on the methodology section. The original paper states: "In SWE-Search, we limit each node to a maximum of three expansions and cap the total search iterations at 100." This implies that generating an answer requires at least 100 steps. Additionally, since each node needs to be evaluated by the value model, there are also at least 100 evaluations, leading to a minimum of 200 inference steps. However, the actual number of inferences is likely much higher because a new iteration involves generating more than just one new node.

D CoT Generation Prompt

Cot Generation System Prompt

You are an expert software engineer and seasoned code reviewer, specializing in bug localization and code optimization within real-world code repositories. Your role is to meticulously analyze code and provide clear, logical reasoning to guide the resolution of issues within the codebase.
In this role, you focus on the precision and effectiveness of the problem-solving process. Your expertise includes understanding complex code structures and accurately mapping issues to the specific parts of the code requiring modification. You excel at breaking down the reasoning process into coherent, easy-to-follow steps that lead to efficient and accurate code fixes.
In this task, we are training a model to generate code modifications for resolving issues within real-world codebases. For this, we have the issue description, the codebase, and the corresponding oracle code modifications. Your task is to generate detailed reasoning to aid in collecting high-quality training data. Your reasoning process must be thorough, evidence-based, and strictly adhere to the provided issue.
Although oracle code modifications are available, **you should simulate reasoning independently—as if you are identifying the necessary files and changes without prior knowledge of those modifications**. Avoid statements like "The edited code makes sense because. . . " that imply direct knowledge of the oracle modifications.

Cot Generation User Prompt

Issue Statement: {problem_statement}
File Content to be Modified: You are provided with the files that require modification to resolve the issue. This includes the full file content. You should identify the code snippets to be modified based on the issue and the file content. {content}
Oracle Code Modifications: {target}
Task Objective: Your objective is to develop a clear and logical reasoning process that guides the modification of the code snippets based on the issue. The reasoning should explain the relationship between the issue and each code snippet, and why the modifications are necessary.
Reasoning Process Guidelines: The reasoning process should generally include the following steps. You may adjust these steps as needed for clarity and accuracy:
1. **Issue Analysis**: - Begin by **clearly articulating the issue**. Provide a comprehensive explanation of why this issue is significant, highlighting the specific challenges or obstacles that must be addressed. Identify the key requirements or objectives necessary for resolving the issue, ensuring that all aspects of the issue are thoroughly examined and understood.
2. **Task Decomposition*: - Break down the overall issue into **smaller, manageable sub-tasks**. Explain the purpose of each sub-task and its significance in solving the issue. Ensure that sub-tasks are logically ordered and clearly connected.
3. **Code Localization and Editing**: - First, for each sub-task, identify the relevant **code snippet** by providing the file path and referring to the specific part of the code related to that sub-task. Next, give a detailed explanation of how this code snippet is connected to the sub-task, explain how the code should be edited to resolve the issue and justify why these changes are necessary. Finally, provide the edited code based on the explanation. - Ensure that the final output for this part MATCHES the provided oracle modifications EXACTLY.
General Requirements:
1. **Clear and Evidence-Based Reasoning**: Provide clear and precise reasoning for each step, strictly based on the provided issue and code without inferring information not explicitly stated. 2. **Comprehensive and Concise**: Address all relevant aspects of the issue comprehensively while being concise. Justify the exclusion of any sections that are not relevant. 3. **Detailed Guidance**: Ensure the reasoning steps are detailed enough to allow someone unfamiliar with the solution to infer and implement the necessary code modifications. 4. **Faithfulness**: Ensure that your final output for the code modifications MATCHES the provided oracle modifications EXACTLY. 5. **Neutral Perspective**: Approach the issue as if you do not know the correct answer in advance. Avoid language that implies prior knowledge of the correct modifications.
Format Requirements:
1. **File path**: Always mention the file path when referring to a code snippet (including class or function names). 2. **Reasoning Process Format**: Use markdown to present your reasoning process. Clearly define each step and ensure logical connections between them. 3. **Code Snippet**: You must include **line numbers** when referring to the original code for context and outputing ‘code_snippet_to_be_modified‘. However, do **not include line numbers** in your editing suggestions.
Please ensure your response is clearly formatted and provides enough detail to justify why each code section was selected for modification and how it should be edited.

E FILE DOCUMENTATION

File documentation

```
{
    "file_path": "examples/bert/bert_model.py",
    "module_docstring": "Bert model ...",
    "classes": [
        {
            "name": "TransformerEncoderBlock(keras.layers.Layer)",
            "docstring": "TransformerEncoderBlock layer. This layer ...",
            "methods": [
                "__init__(self, num_attention_heads ...)",
                "build(self, input_shape)",
                ...
            ]
        }
        ...
    ],
    "functions": [
        {
            "name": "make_attention_mask(inputs, mask)",
            "content": "def make_attention_mask(inputs, mask):
            [First five lines]
             ...
            [Last five lines]"
        }
        ...
   ]
}
```

Figure 5: Example of a file documentation. The documentation includes the relative file path and module docstring (if available). It also contains class names, their associated docstrings, and all method names. For functions, only the name and the first/last five lines of code are included.

F EDIT OUTPUT PATCH EXAMPLE

```
{
  "edited code": [
    {
      "file": "src/flask/blueprints.py",
      "code snippet to be modified": "
          188         template_folder=template_folder,
          189         root_path=root_path,
          190       )
          191       self.name = name
          192       self.url_prefix = url_prefix
          193       self.subdomain = subdomain",
      "edited code snippet": "
                        template_folder=template_folder,
                        root_path=root_path,
                      )

                      if "." in name:
                        raise ValueError("'name' may not contain a dot '.' character.")

                      self.name = name
                      self.url_prefix = url_prefix
                      self.subdomain = subdomain"
    },
    {
      "file": "src/flask/blueprints.py",
      "code snippet to be modified": "
          360       """Like :meth:`Flask.add_url_rule` but for a blueprint.  The endpoint for
          361       the :func:`url_for` function is prefixed with the name of the blueprint.
          362       """
          363       if endpoint:
          364         assert "." not in endpoint, "Blueprint endpoints should not contain dots"
          365       if view_func and hasattr(view_func, "__name__"):
          366         assert (
          367           "." not in view_func.__name__
          368         ), "Blueprint view function name should not contain dots"
          369       self.record(lambda s: s.add_url_rule(rule, endpoint, view_func, **options))
          370
          371     def app_template_filter(self, name: t.Optional[str] = None) -> t.Callable:",
      "edited code snippet": "
                      """Like :meth:`Flask.add_url_rule` but for a blueprint.  The endpoint for
                      the :func:`url_for` function is prefixed with the name of the blueprint.
                      """
                      if endpoint and "." in endpoint:
                        raise ValueError("'endpoint' may not contain a dot '.' character.")

                      if view_func and hasattr(view_func, "__name__") and "." in view_func.__name__:
                        raise ValueError("'view_func' name may not contain a dot '.' character.")

                      self.record(lambda s: s.add_url_rule(rule, endpoint, view_func, **options))

                    def app_template_filter(self, name: t.Optional[str] = None) -> t.Callable:"
    }
  ]
}
```

G CASE STUDY

We perform an in-depth analysis of several representative special cases and identify key issues in the model's predictions. In some instances, the model's output deviates from the standard code patch but still provides reasonable edits based on the retrieved results, ultimately resolving the problem effectively.

For instance, in the case of *psf__requests-2674*, the standard solution identifies the issue in the file *requests/adapter.py*, while our model retrieves *requests/expectation.py*. According to the standard evaluation, the model's prediction would be considered incorrect. However, the patch generated

during the editing stage successfully resolves the underlying issue. This illustrates that, despite differing from the standard solution, the model's approach is both valid and effective.

This observation highlights a critical limitation of evaluation methods that rely solely on rule-based matching. Such methods fail to account for the diversity of valid solutions, potentially overlooking unique and effective editing strategies. To address this, future research should prioritize the development of real code execution environments to evaluate repair effectiveness through actual execution, rather than relying exclusively on static comparisons with standard answers.